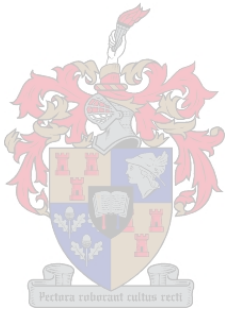
A Cross Platform Framework for Software Defined Radio

Richard Brady



Thesis presented in partial fulfilment of the requirements for the degree
Master of Science in Electronic Engineering
at the University of Stellenbosch

Supervisor: Dr G-J van Rooyen

March 2007

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature

Date

Abstract

Software defined radios (SDRs) implement in software those parts of a radio which have traditionally been implemented in analogue hardware. We explain the importance of this definition and introduce reconfigurability and portability as two further goals. Reconfigurabilty is a property of the SDR platform, which may be a microprocessor, configurable hardware device, or combination of the two. We demonstrate that the field-programmable gate array is sufficient for the implementation of practical SDR systems. Portability, on the other hand, is a property of the modulation and demodulation software, also known as waveform specification software. We evaluate techniques for achieving portability and show that waveforms can be specified in a generic form suitable for the autogeneration of implementations targetting both microprocessor- and FPGA-based architectures. The generated code is in C++ and VHDL respectively, and the tools used include formal models of computation and the XSLT language.

Opsomming

In 'n sagteware-gedefinieerde radio (SDR) word radio-komponente wat tradisioneel in hardeware geïmplementeer word, se funksionaliteit deur sagteware vervang. Ons beklemtoon die belangrikheid van hierdie definisie, en stel dan herkonfigureerbaarheid en oordraagbaarheid voor as verdere doelwitte. Herkonfigureerbaarheid is 'n eienskap van die SDR-platform, wat 'n mikroverwerker, konfigureerbare apparatuur, of 'n kombinasie van die twee kan wees. Ons demonstreer dat 'n FPGA geskik is vir die implementering van praktiese SDR-stelsels. In teenstelling met herkonfigureerbaarheid is oordraagbaarheid 'n eienskap van die modulasie- en demodulasie-sagteware – ook bekend as die golfvorm-spesifikasiesagteware. Ons evalueer tegnieke om oordraagbaarheid te bewerkstellig, en toon aan dat golfvorms generies gespesifiseer kan word, in 'n vorm wat outomatiese kodegenerasie toelaat, met beide mikroverwerker- en FPGA-gebaseerde stelsels as teikens. Kode word onderskeidelik in C++ en VHDL gegenereer, en berus op formele modelle van berekening en die XSLT taal.

Terms of Reference

This project was commissioned by Dr Gert-Jan van Rooyen of the Department of Electrical and Electronic Engineering at the University of Stellenbosch. His specific instructions were:

- To evaluate the roles of reconfigurability and portability in the field of software defined radio.
- To develop platform independent techniques for specifying digital signal processing algorithms to modulate and demodulate radio waveforms.
- To evaluate field-programmable gate array (FPGA) architectures as implementation platforms for modulation and demodulation algorithms.
- To design a VHDL framework for implementing modulation and demodulation algorithms on FPGAs, to complement *libsdr*, the existing framework in C++.
- To design mechanisms for the autogeneration of both VHDL and C++ from a single top-level specifcation of a modulation or demodulation algorithm.
- To provide proofs of concept for any proposed systems.

Acknowledgements

I would like to thank:

- my supervisor, Gert-Jan van Rooyen, for his guidance,
- Ed Willink of Thales Research in the UK, for his interesting ideas,
- my girlfriend, family and friends, for their support.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

ADC	analogue-to-digital converter
AM	amplitude modulation
API	application programming interface
ASIC	application specific integrated circuit
AST	abstract syntax tree
CPU	central processing unit
CT	continuous time
DAC	digital-to-analogue converter
DDS	direct digital synthesis
DLL	delay-locked loop
DPLL	digital phase-locked loop
DSL	domain specific language
DSP	digital signal processing
DT	discrete time
EEPROM	electronically erasable programmable read-only memory
FIFO	first-in first-out
FIR	finite impulse response
FM	frequency modulation
FPGA	field programmable gate array
FPU	floating-point unit
FSK	frequency shift keying
GPP	general purpose processor
JTRS	Joint Tactical Radio Program
LAB	logic array block
LE	logic element
LPF	low-pass filter
LUT	look-up table
MAC	multiply and accumulate

MVC	model view controller
NCO	numerically controlled oscillator
OMG	Object Management Group
OO	object-oriented
OS	operating system
OSI	Open Systems Interconnect
PLD	programmable logic device
PLL	phase-locked loop
PTT	push-to-talk
QBBDDS	quadrature baseband direct digital synthesis
RAM	random-access memory
RDL	Radio Description Language
RFE	radio front-end
RISC	reduced instruction-set architecture
RTL	register transfer level
SAC	Single Assignment C
SCA	Software Communication Architecture
SDF	synchronous dataflow
SDR	software defined radio
SDRF	SDR Forum
SLOC	source lines of of code
SNR	singal-to-noise ratio
SRAM	static random-access memory
SU	Stellenbosch University
UML	Unified Modeling Language
VCO	voltage-controlled oscillator
VHDL	VHSIC Hardware Description Language
VHSIC	very high-speed integrated circuit
VM	virtual machine
WDL	Waveform Description Language
XML	eXtensible Markup Language
XSD	XML Schema Definition
XSL	XML Stylesheet Language
XSLT	XSLT for Transformations

Chapter 1

Introduction

Software defined radio (SDR) systems use hardware and software technology to reduce the amount of analogue signal processing in radio applications. They do this by implementing the conversion between analogue and digital signals as close to the antennae as possible, and by then performing signal processing operations (which would typically have been performed by analogue components in a hardware radio) in the software domain. This architecture is advantageous because of its flexibility, as it allows a single hardware platform to be reused for different applications, and reduces system upgrades to software updates [2].

This technology has developed rapidly in recent years due to the exponential rise in the speeds at which data conversion (between analogue and digital signals) and computation can be performed. As a result, the concept has found broad commercial application, with uses ranging from commercial radio broadcast to cellular base stations [13].

While the principles of SDR apply similarly across all applications, the hardware (both analogue and digital) can be somewhat diverse. Of particular interest here, are the several device categories which are capable of performing the software domain processing. These may be application-specific integrated circuits, but are more likely to be generic software platforms. They include, amongst others, digital signal processors (DSPs), general purpose processors (GPPs) and field programmable gate-arrays (FPGAs). The choice of platform is influenced by factors such as performance, cost and time-to-market.

The software which is installed and runs on these platforms is central to the design of an SDR system, and advanced software architecture and development techniques have been applied to the problem [37]. Modular software design allows the reuse of code which is common to more than one application. For example, a phase-locked loop could be implemented as a software module, and could be reused in several different demodulation applications, since many receiver systems include this basic component.

The University of Stellenbosch has developed an SDR system (SU SDR) with software capable of modulation, demodulation and several other digital signal processing operations. The majority of the system is implemented in the C++ programming language. It supports

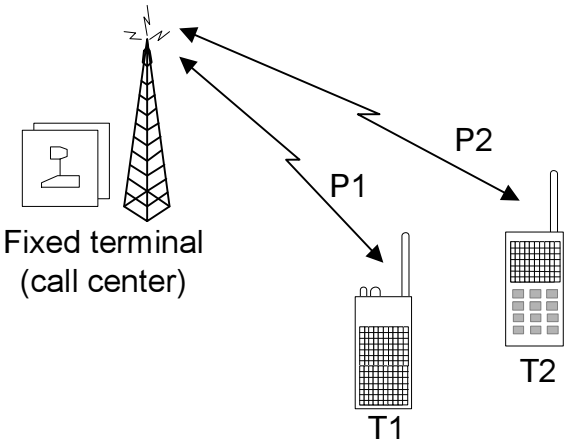


Figure 1.1: *Two-way radio systems with interoperability problems.*

dynamic reconfiguration, and has been successfully interfaced with radio front-end (RFE) hardware [20]. Development is ongoing.

There is a clear need for a framework under which functional and architectural descriptions can be separated from platform-specific features, while supporting multiple platforms as implementation targets. The aim of this research has been to investigate this need by taking all of the above into consideration.

In order to demonstate the utility of such a cross platform framework, a practical example is provided next.

In two-way radio systems, compatibility between different handsets on different protocols is an ongoing problem. This lack of interoperability is especially problematic for emergency services, with different teams (e.g. the fire brigade and the traffic department) often unable to communicate at the scene of an emergency. Figure 1.1 demonstates the topology of such a system. In this example, two different departments having handset types T1 and T2 are communicating on protocols P1 and P2 respectively. Because P1 and P2 are not compatible, communication between the departments is not possible. In the case where personnel are fortunate enough to be within range of a base station which supports both protocols P1 and P2, messages may be relayed by an operator, but this is a cumbersome means of communication which introduces a significant bottleneck.

By installing an SDR base station, support of protocols P1 and P2 would simply require the presence of two software modules, one supporting each protocol. An SDR handset could be designed on the same principle. Most significantly, if the software for protocols P1 and P2 is written in a manner which is sufficiently portable, the handset and base station could be built using the same source code while relying on different hardware platforms. For example, the base station might be implemented on a standard workstation computer while

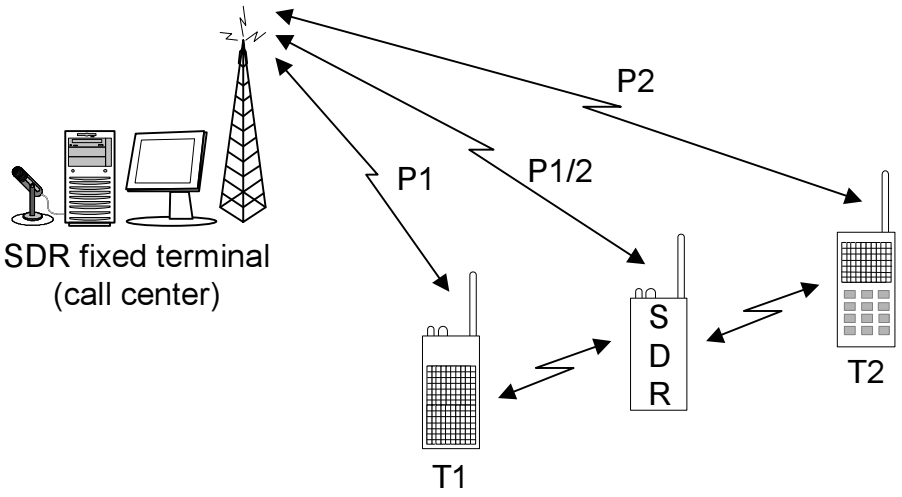


Figure 1.2: *Two-way radio system with SDR components.*

the handset is implemented on an FPGA-based platform.

In order to realise the advantages of SDR which have been highlighted by this example, this thesis aims to:

- find a common radio description which is platform-independent,
- design platform-specific architectures for the target platforms, and
- develop the necessary translation mechanisms to convert from radio description to implementation.

We begin our analysis by providing a background in Chapter 2, where we discuss the concept of SDR in more detail and consider the technical aspects of FPGA and microprocessor (μP) platforms. In Chapter 3 we introduce a reference waveform and an FPGA-based reference platform to aid in the discussion throughout the document, and provide a means for evaluating the usefulness of our solutions.

In Chapter 4 we develop a system for the specification of waveforms using a domain-specific language (DSL) based on XML. In Chapter 5 we design a VHDL framework for the implementation of these specifcations on FPGAs. We then bring these concepts together in Chapter 6 by showing techniques for automatically generating waveform implementations from specifications.

In Chapter 7 we evaluate the above by applying the process to the reference waveform introduced in Chapter 3 and demonstrating the correct demodulation of practical signals. Concluding remarks are given in Chapter 8.

Chapter 2

Background

In this chapter the field of software defined radio is discussed. The integration of software and radio components to form useful systems is described, and boundaries which separate SDR from other digital or software-based communication systems are defined. It is demonstrated that reconfigurability and portability are central to the distinction. The different classes of platforms capable of meeting these criteria are considered, along with the means for specifying radio waveforms on each. Differences in these specifcation methods are then highlighted, and the possibility of direct translation is eliminated. The need for an additional layer of abstraction is identified, and the platforms and languages which require support are introduced.

2.1 Software defined radio

Software defined radio (SDR) is a term with several interpretations. At the centre of any definition, however, is the concept of performing in software what would otherwise be performed in analogue hardware. This has become a significant trend in the design and implementation of radio transceivers [37], and involves placing analogue-to-digital converters (ADCs) and digital-to-analogue converters (DACs) as close to the antenna as is reasonable and possible (see Figure 2.1).

ADCs and DACs (also known as data converters) are absent in most traditional receivers, where integrated circuits (ICs) are responsible for the modulation and demodulation of the waveform, as depicted in Figure 2.2. The distinction between software and hardware defined radios is not always this clear [13]. When assessing whether or not a system qualifies as an SDR, the following must be taken into consideration.

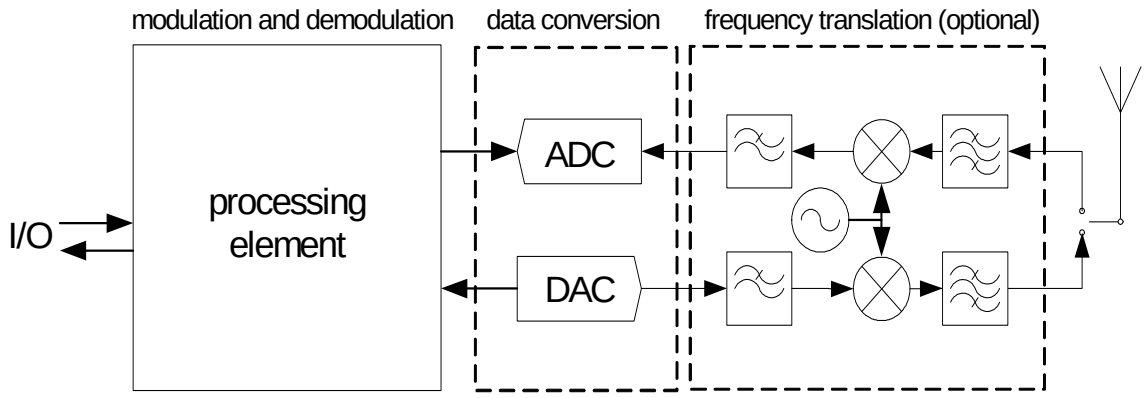


Figure 2.1: *Software defined radio architecture.*

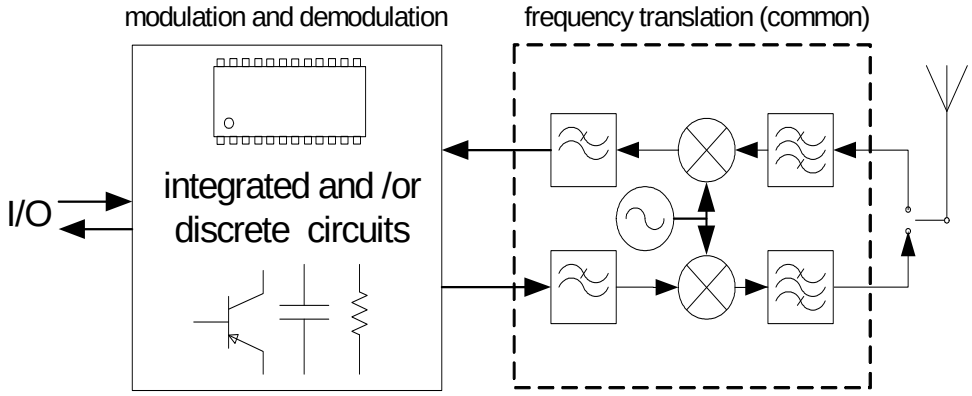


Figure 2.2: *Traditional radio architecture.*

2.2 OSI layer

Software has long been used in communication systems. Any networked computer has a substantial portion of the networking protocol stack implemented in pure software running on the generic CPU of the host machine. Additionally, the analogue signals in such networks, such as the 10BASE-T electrical standard, are electromagnetic in nature and operate at frequencies well into the frequency range associated with radio. This leads one to wonder whether such systems are not effectively software defined radios.

One possible answer to this question lies in the word *radio*, which refers specifically to the *radiation*, or wireless transmission, of electromagnetic energy. The use of open air (and sometimes other mediums such as the ocean) with radiation creates a shared medium, which we ration by frequency-, time- and code-division. The absence of cables also enables the mobility of terminals, and a mobile terminal must constantly adapt to perform in its current location. Wired terminals on the other hand are usually fixed in location and therefore not

Table 2.1: *The seven layers of the OSI Reference Model.*

	Layer	Function
7	Application	Network process to application
6	Presentation	Data representation and encryption
5	Session	Interhost communication
4	Transport	End-to-end connections and reliability
3	Network	Path determination and logical addressing
2	Data link	Physical addressing
1	Physical	Media, signal and binary transmission

subject to this requirement. Furthermore, the signals are baseband in nature, and do not require the advanced filtering and mixing which is used for the extraction of passband singals from radio channels. These differences mean that solutions for wired terminals are not easily generalised to the wireless case, and so software defined networking is not equivalent to software defined radio.

Another answer lies in the Open Systems Interconnection Reference Model (OSI Model for short) [49], which is depicted in Table 2.1. The model provides a classification mechanism for the various components and protocols of communication systems, both wireline and wireless. The first, or physical layer is often implemented in analogue hardware, while layers two through seven are usually implemented in a combination of firmware and software. The physical layer is responsible for modulation and demodulation, and the channel medium is often a wireless or radio link. Therefore a software implementation of the physical layer may be classified as SDR, but this is not the case with higher layers.

Of course not all systems conform to the OSI model. A push-to-talk (PTT) handheld FM radio is far simpler, but is also a candidate for SDR. We resolve this by considering such terminals as special cases of the OSI model consisting of only a physical layer. The simplicity of these devices has tremendous value in exploring the field of SDR.

2.3 Reconfigurability

Reconfigurability is central to the concept of software defined radio. It is commonly considered to be the criterion by which SDR is separated from *digital radio* [13], where transceivers use digital signal processing but are no more configurable than an analogue implementation. In devices with firmware components performing radio functionality, the firmware must not only be easily reprogrammable, but must also be sufficiently powerful for such a reprogram-

ming to result in support for a different waveform.

A *waveform* or *air interface* is a mapping from information to a signal, most easily described in terms of a modulation and demodulation scheme. Depending on whether the communication channel is simplex or duplex, either or both of modulation and demodulation must be supported. There are countless such waveforms, all of which have been designed with a specific set of applications mind. A device which can support more than a single waveform, but using the same hardware and only software changes to switch between supported waveforms, is a software defined radio.

This concept of reconfigurability is not a binary property, but a measure which can fall anywhere between extremes. At one end of the scale, we have devices with set functionality. A radio frequency identification (RFID) tag is an example. These small radio devices are integrated into a single package with no external connections. They support a single waveform at a fixed frequency, and are not reconfigurable or tunable in any way.

Next are radios with parameters which can be adjusted, such as frequency (or station number) and output volume in a commercial broadcast radio receiver. Transceivers which support more than one waveform by implementing hardware for both and simply switching between circuits, also fall under this category. For example, many commercial radio receivers support both AM and FM in this way.

In [19] Bose *et. al.* make the important distinction between *known waveform* and *new waveform* reconfigurability. In the former, only waveforms considered during platform design can be supported. In the latter, waveforms which are significantly distinct from those considered at the time of platform design can be supported. The extreme case of new waveform reconfigurability, the ability to support all possible current and future waveforms, is a currently unattainable ideal [13], but remains a good point of reference.

With the signal of interest in a digital form, and a sufficiently advanced computing platform, many of the powerful and established techniques of software engineering can be applied to the problem. This has been the subject of much research [12, 15, 18, 20], and it has been shown that principles such as object-oriented design are powerful in the context of SDR.

2.4 Portability

While it is essential that platforms support multiple waveforms, it is also highly desirable for waveforms to support multiple platforms. Support for a waveform consists of an executable algorithm for modulation, demodulation or both. The execution of this algorithm must occur at the rate required by the application, which may tolerate latency but almost always demands on-line processing, the ability to process the data at a rate equal to, or greater

than, the rate of arrival. This requirement is a primary criterion in separating simulation from implementation, since the two can become difficult to distinguish in software defined digital signal processing systems.

Because of the performance requirements mentioned above, it is common for waveforms to be specified in a language which is optimised for the target platform. This was the case with the SpeakEasy project launched in 1991 by the United states Department of Defence. The project aimed to support at least ten military waveforms on a single platform [14]. That platform was based on the Texas Instruments TMS320C40, the fastest DSP device at the time. More than one device was required due to the computational requirements of the system. The software was developed and optimised specifically for this device to reduce the number required. By the time of completion of Phase 1 of the project in 1994, Texas Instruments had released the four times more powerful TMS320C54 processor. The two device families were, however, not code compatible, meaning that the SpeakEasy project could not port the waveforms to the new device without a full rewrite of their code [18].

The vast improvement in processor technology which occurred while the SpeakEasy system was under development was not unique to that time or to their application. Instead, it is a continuing effect often described as Moore's law, which observes that transistor density on integrated circuits tends to double every 18 to 24 months. This has the counter-intuitive implication that, over time, portable software will frequently outperform software which has been optimised for a specific platform [18].

For this reason, it is common for waveforms to be written in high-level programming languages, and then compiled separately to execute on specific platforms. With a portable language such as C, compilers exist for a variety of targets, and so a waveform described in (suitably generic) C may be targeted to an x86 architecture or a DSP chip. These are distinct targets, the x86 complying with the Von Neumann architecture while most DSP chips are based on a Harvard architecture [42] and have radically different instruction sets.

However, both targets are microprocessors, executing instructions on data in a sequential fashion. Against the backdrop of all computing devices available, this makes them fairly similar. There exist far more distinct devices with entirely different architectures which must be explored.

The field-programmable gate-array (FPGA) [41] is one of these (see Section 2.5.3). In a similar way to that described above, a radio algorithm may be specified in the high level VHDL (see Section 2.5.4). This would allow the compilation of waveforms to target a variety of FPGA devices from different vendors, each with a unique internal architecture. In fact, VHDL is also a suitable design entry language for other classes of devices, such as application specific integrated circuits (ASICs) [41].

In both of the examples given above, the waveform itself can be compiled to more than

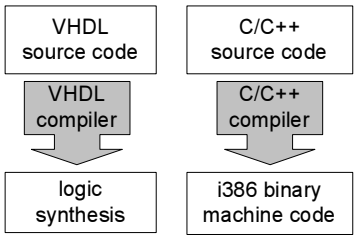


Figure 2.3: *Conventional software development.*

one target and is therefore portable to some extent. Cross compilation on the other hand, is far less simple. It is extremely difficult to compile C code so that it executes efficiently on an FPGA, and equally challenging to compile VHDL so that it executes well on a microprocessor. We are therefore left with two separate design paths, as depicted in Figure 2.3.

This is an undesirable arrangement, since software modules which are required to operate on different platforms must be developed in one language, and then rewritten into other languages, as code reuse via direct translation is not practical.

The problem with translation lies in the differing semantics of the various high-level languages. These differences are sometimes superficial, such those between C and Java for example. However, the semantic differences between C and VHDL are vast. This stems from their differing *models of computation* (see Chapters 4 and 5). In a microprocessor (μP) based platform, such as a GPP or DSP, instructions are executed in sequence, and this is reflected in the imperative, or procedural, programming paradigm of C.

In FPGA platforms, several parts of the device can perform computations concurrently, which is reflected by the declarative, or functional, programming paradigm of VHDL. The VHDL language includes imperative mechanisms (see Section 2.5.4) but these are not sufficient to enable direct translation.

The translation challenge can be simplified by selecting a subset of one language which pairs well with a subset of the other in terms of functionality. The Single Assignment C (SAC) [25] subset of C has been popular in this regard, and compilers have been implemented [26, 22]. However, diverse languages exist exactly because each is powerful in a unique way, and so their best features seldom overlap. As a result, the subset approach often discards powerful properties of both languages. For example, pointers cannot be supported inside an FPGA, but are at the core of any μP-based DSP system.

An alternative approach is possible. By applying a further layer of abstraction to the specification process, platform-specific characteristics can be hidden, and the software description can focus on functionality alone. This abstraction would be implemented as an even

higher-level language describing algorithms alone, with compilers to generate each language on the level below. In order to hide the peculiarities of each language while still being able to use them in the compiler output, we look for dominant features in our applications and ask how they map to each target language. We turn these features into constructs of the higher-level language, and allow our compilers to recognise those constructs and generate the best implementation for each target.

We cannot identify the dominant features for all classes of programs, and so must limit our higher-level language to a specific set of tasks. Such languages are known as domain specific languages (DSLs) [30] and are common in the software community. DSLs for software defined radio have been proposed [18, 47] and will be discussed in Chapter 4.

The implementation of such a system, called *libsdr*, is underway at the Stellenbosch University SDR project (SU SDR) [4]. The abstract language used is the Extensible Markup Language (XML). A cross-compiler then uses the Extensible Stylesheet language (XSL) to specify transformations which generate source code similar to that depicted in Figure 2.3. XSL files for different platforms describe how the XML functionality can be achieved on those platforms. Figure 2.4 summarises this concept, and shows the existing implementation.

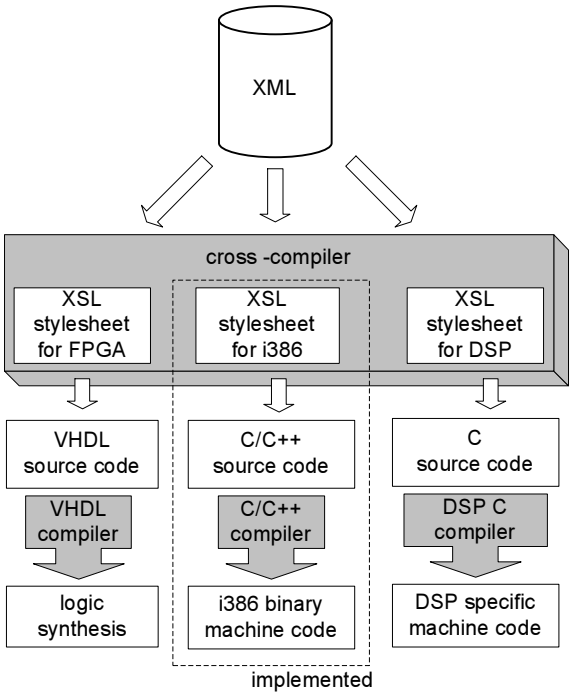


Figure 2.4: *Additional abstraction layer for software design.*

This concept of waveform portability is the third of three criteria which we have defined for distinguishing software defined radios. However, portability is not a strict requirement. The SpeakEasy project described above is an example of an SDR system which did not

support portability. It is, however, a highly desirable property, and as such warrants the investigation which constitutes this thesis.

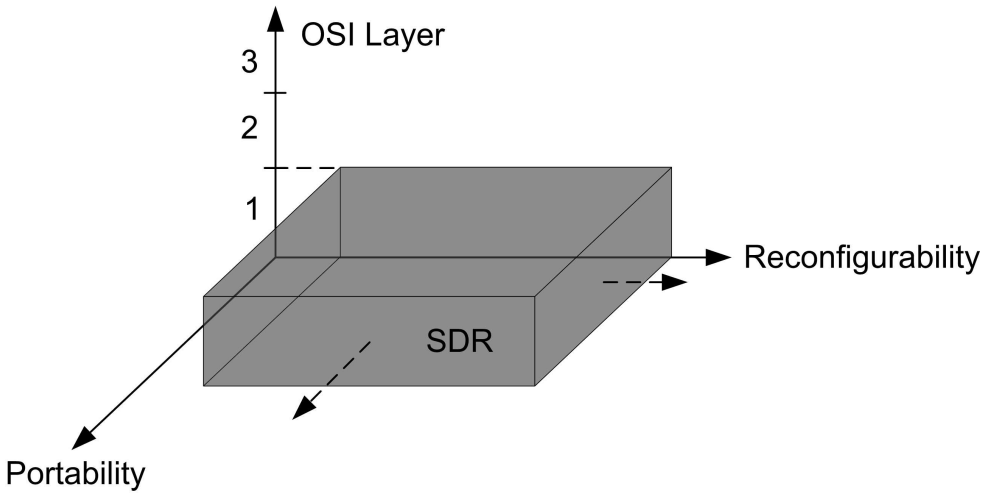


Figure 2.5: *Classification as SDR.*

Because reconfigurability, portability, and classification in terms of the OSI Model are independent aspects of a radio, they can be visualised as orthogonal axes in the space of all radio systems, as shown in Figure 2.5.

2.5 Target platforms

The primary goal of this study is the compilation of modulation and demodulation algorithms for execution on multiple platforms. These platforms each require a unique program model, and therefore unique programming languages for design capture. In this section, platforms based on two particular device categories are discussed, along with the languages most commonly used for digital signal processing on each. They are microprocessor (μP) and field-programmable gate array devices, and are most commonly used with C++ and VHDL, respectively, for DSP applications.

2.5.1 μP-based platforms

Microprocessor-based platforms implementing SDR make use of wideband ADCs and DACs in combination with a central processing unit (CPU), which performs modulation or demodulation, and several other devices, including memory and high-speed buses. Two classes of CPUs are common. Digital signal processors (DSPs) are optimised for arithmetic operations which are typical of SDR and similar applications. They run without an operating system,

and are most often programmed in either a low level assembly language or C. General purpose processors (GPPs) on the other hand are targeted at a broader range of applications, most notably the workstation computer. The high demand and strong competition in the GPP market cause these devices to develop at a rapid pace while often remaining backward compatible with previous generations. They are usually used in conjunction with an operating system (OS).

The GPP-based workstation computer provides an attractive platform for SDR applications. Notable implementations include the SU SDR project [4] and the open source GNU Radio project [1]. Bose *et al.* [15] refer to this as the *virtual radio* platform, and argue that workstations such as the personal computer offer several advantages, including:

- ease of experimentation due to programmer familiarity with the platform,
- rapid deployment based on existing software deployment techniques,
- integration with other applications running on the same workstation,
- reduced cost when developing low-quantity (specialised) products and when prototyping, and
- dynamic reconfiguration using existing operating system features such as dynamically loaded libraries (DLLs).

2.5.2 C++ language

A significant advantage of μP-based platforms is the wide range of software and development tools available, which includes many compilers supporting various high-level languages. Of particular interest here is C++, a high-level language supporting the object-oriented [23] paradigm. When compiled, the resulting machine code exhibits strong performance in DSP applications [20].

C++ is currently used by the SU SDR project as an implementation language. An object-oriented framework has been designed, in the form of a class library, to allow modular programming of SDR systems (see Chapter 5). The current XML-based cross compiler of Figure 2.4 generates C++ which uses this framework, and shall be discussed in Chapter 6. Language features used by the framework include inheritance, which is used by defining generic *converters* (a DSP block with inputs and outputs) upon which users can base specialised DSP blocks, and error handling in the form of exceptions.

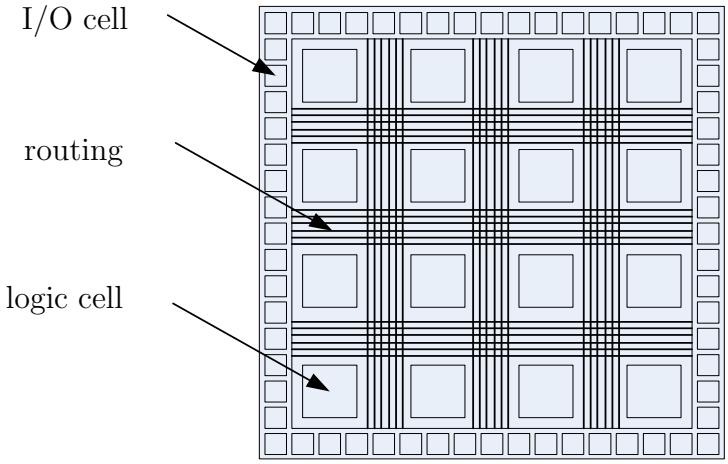


Figure 2.6: *Simplified diagram of a generic FPGA.*

2.5.3 FPGA-based platforms

An FPGA is a configurable integrated circuit [41], consisting of large arrays of programmable logic elements, interconnected by many programmable routing signals, as demonstrated in Figure 2.6. The logic elements vary in complexity across vendor and device, but usually contain primitives such as AND, OR, XOR, NOT, or small look-up tables, as well as memory elements such as latches and flip-flops. Modern FPGAs may also include larger but more specialised components, such as RAM or dedicated multiply and accumulate (MAC) blocks. By configuring the logic operations and interconnections, the device can be made to behave as desired.

A range of technologies are used in different FPGA designs to make them configurable [41]. Antifuse technology uses the passing of high currents (approx. 5 to 15 mA) during programming to permanently melt non-conducting materials within the device so that they become conductive links. Antifuse devices may be programmed only once, but do not require the continued application of power to retain their configuration. In EEPROM (electronically erasable programmable read-only memory) and flash memory based devices, configuration is also non-volatile, but can can be repeated.

Of particular interest are static RAM (SRAM) based devices. These devices are also reconfigurable and use an SRAM layer inside the device to hold configuration data. SRAM is volatile, meaning that the device must be reprogrammed every time power is applied, but the advanced nature of SRAM fabrication technology permits high gate densities, making these attractive targets for arithmetic intensive applications such as SDR [21].

To overcome the problem of volatility, SRAM-based FPGAs are usually accompanied on-board by a small configuration controller, such as a programmable logic device (PLD), and

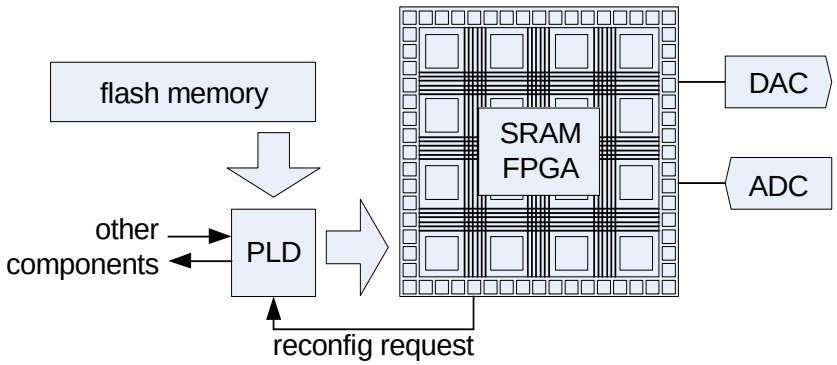


Figure 2.7: *Configuration system for SRAM-based FPGA.*

non-volatile memory (such as flash memory) to store the configuration. The configuration controller is responsible for loading configuration data from memory into the device at power-up or when a reprogramming is requested, as shown in Figure 2.7. Despite the added circuit complexity, this arrangement can be advantageous. A new configuration can be transferred at any data rate to the flash memory, and once there, can be used to reconfigure the device in a matter of milliseconds.

FPGA-based platforms are no match for the virtual radio in terms of reconfigurability, but assuming that waveform support is implemented inside the FPGA, this arrangement does meet the minimum criterion discussed in Section 2.3. For this reason we state the FPGAs are sufficiently reconfigurable to act as platforms for SDR.

Additional advantages of FPGA-based platforms include:

- high data throughput and speed of computation due to truly parallel execution,
- low power consumption, and
- low level access to input and output devices such as data converters.

2.5.4 VHDL

The FPGA falls under a broader device category known as very-high-speed integrated circuit (VHSIC) devices. A language in common use for the modelling of such devices is VHSIC Hardware Description Language, or VHDL. VHDL differs significantly from other common programming languages in that it has event-driven semantics. This requires a globally consistent notion of time, a feature not present in procedural programming languages such as C++. The VHDL language specification [31] specifies how time evolves during execution, and we refer the reader to Rushton [39] for a summary.

Listing 2.1: *VHDL for an OR operation.*

```
library ieee;
use ieee.std_logic_1164.all;

entity or_gate1 is
    port (
    in1, in2 : in std_logic;
    out1     : out std_logic
    );
end or_gate1;

architecture behaviour of or_gate1 is
begin
        out1 <= in1 or in2;
end behaviour;
```

While VHDL was not originally intended for use as a design entry language for logic synthesis, a subset [17] was identified for this purpose and has found widespread use. Software programs called *synthesis tools* take VHDL as input and generate configuration data for the device as output. This subset of VHDL is commonly referred to as register transfer level (RTL) VHDL [39], because compliant code can be reduced to an equivalent form consisting of sets of registers connected only by combinatorial (memoryless) logic.

Synthesis tools look for common constructs which map to registers or combinatorial logic (template matching) and convert these to gates and registers. Listing 2.1 shows an example of VHDL code and Figure 2.8 the equivalent logic circuit. The top level constructs in VHDL are the *entity*, which defines an interface, and the *architecture* which implements an interface.

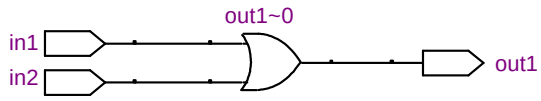


Figure 2.8: *Synthesis results for Listing 2.1.*

An architecture has a declaration part, where local signals (as well as other constructs such as constants and functions) are declared, and a body, where relations between inputs, outputs and signals are defined in a declarative manner. These relations may be combinatorial as in Listing 2.1, where the target of the assignment is updated if any input or signal in the expression changes, or they may be specified in way that infers memory. This is done using the *process* construct, shown in Listing 2.2 which has three important features. First,

Listing 2.2: *VHDL for an OR operation with a register.*

```
library ieee;
use ieee.std_logic_1164.all;

entity or_gate2 is
    port (
    clock    : in std_logic;
    in1, in2 : in std_logic;
    out1     : out std_logic
    );
end or_gate2;

architecture behaviour of or_gate2 is
begin
    process (clock)
        begin
            if rising_edge(clock) then
                out1 <= in1 or in2;
            end if;
        end process;
end behaviour;
```

it may be given a sensitivity list, which is a list of only those signals whose changes it must respond to. In synchronous designs the only such signal is the clock. Second, it permits the declaration of variables, which retain their value between successive invocations of the process (in response the events on signals in the sensitivity list). Third, the body of the process escapes the declarative nature of the architecture body, and takes on an imperative style, allowing variables to be assigned to more than once, and the order of statements to thus have meaning.

Examples of VHDL files with and without a process are given in Listings 2.1 and 2.2 respectively. The corresponding synthesis tool outputs are shown in Figure 2.8 which is a simple OR gate, and Figure 2.9 which demonstrates the mechanism of flip-flop register inference when a process is written to respond only to the rising edge of the clock signal.

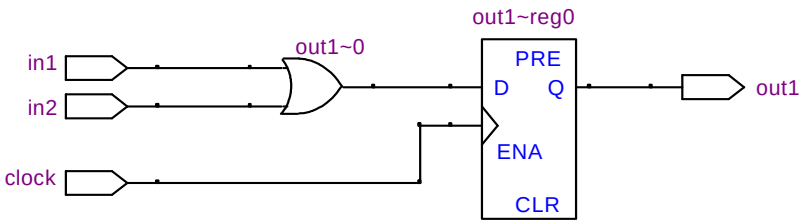


Figure 2.9: *Synthesis results for Listing 2.2.*

Chapter 3

Reference implementation

In Chapter 2 we introduced the concept of waveform portability, where a *waveform* is an algorithm for modulation or demodulation. This terminology is used widely in the field of SDR. A central goal of this thesis is to achieve waveform portability between FPGA- and μP-based platforms, by autogenerating VHDL and C++ code respectively, both from a single specification.

This requires the development of systems for the specification and implementation of waveforms, as well as for the transformation from specification to implementation. Because all three of these topics are very broad research fields on their own, we need to reduce the problem to a goal which is achievable within the scope of this thesis. We therefore introduce a reference waveform and reference platform in this chapter, and make it the goal of this thesis to achieve waveform portability with respect to these specific examples.

3.1 Reference waveform

Frequency modulation (FM) is a common waveform used for the transmission of both analogue and digital signals. Examples include commercial FM radio broadcasting, which is analogue in nature, and frequency shift keying (FSK), for the encoding and transmission of data. As a reference design we consider the case of audio signals transmitted using FM.

In FM, a sinusoid is transmitted of which the frequency varies in proportion to the amplitude of the modulating input signal.

$$y(t) = A\cos\left(2\pi\left[f_c + f_d \cdot x(t)\right]t\right) \tag{3.1}$$

Here $x(t)$ is the message signal and $y(t)$ is the modulated signal, having amplitude A, centre freqency f_c and a peak frequency deviation of f_d.

3.2 Modulation by DDS

In analogue circuits FM is generated with a voltage controlled oscillator (VCO), shown in Figure 3.1 with an equivalent mathematical model in Figure 3.2. An integrator accumulates

phase which is passed to a cosine function in order to obtain the output. The input to the integrator is the result of multiplying the input by some gain, to achieve the desired frequency deviation, and then adding an offset (or bias), to determine the centre frequency.

Figure 3.1: *Voltage controlled oscillator (VCO).*

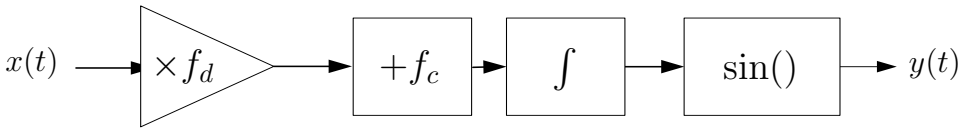


Figure 3.2: *Expanded model for the VCO.*

This model can be implemented digitally, and analogue FM signals can be generated by feeding the digital output to an ADC. This is known as direct digital synthesis (DDS) [43]. A Simulink model of such a system is shown in Figure 3.3. The integrator is implemented as a fixed-point accumulator with maximum and minimum values separated by a multiple of 2π so that wrapping on overflow causes a discontinuity in phase but not in the output of the cosine function which has a period of 2π. The cosine function is implemented as a look-up table, and when combined with the accumulator, is known as a numerically controlled oscillator (NCO).

Because the system is digital, parameters must be normalised by the sampling frequency, F_s, so that they are in the correct units, such as *cylces per sample* instead of *cycles per second*, or hertz.

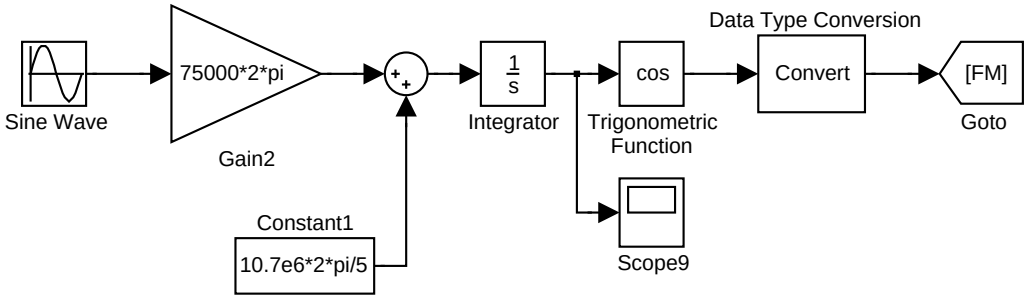


Figure 3.3: *Simulink model for FM modulator.*

3.3 Demodulation by DPLL

Several methods exist for demodulating FM [48]. A system which yields an output proportional to the frequency deviation of the input is known as a discriminator. One possible implementation involves differentiating the signal and passing it through an envelope detector. This works because the derivative of a sinusoid is another sinusoid of the same frequency, but with an amplitude proportional to that frequency.

Another discriminator implementation is the phase-locked loop (PLL). The PLL controls an oscillator with feedback in such a way that its frequency follows that of the input. A mathematical model is shown in Figure 3.4. When the PLL is in lock, the output of the VCO has the same frequency as the input signal, and their phases are therefore offset by a constant amount. This amount is 90 degrees if the VCO's free-running frequency (the frequency of its output for zero input) is equal to the center frequency of the input signal.

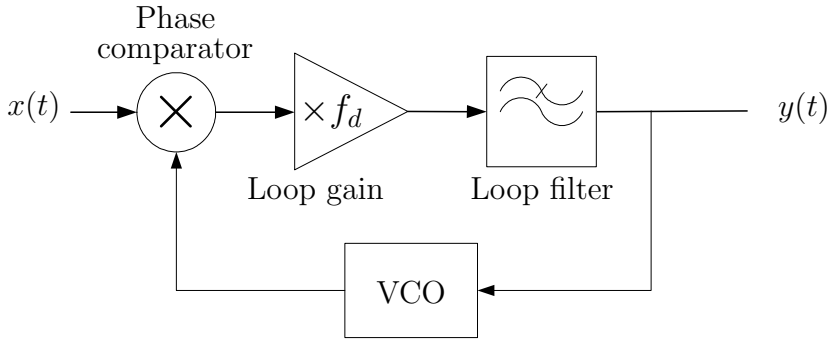


Figure 3.4: *Phase-locked loop.*

The multiplication operation at the input performs the function of a phase comparator when combined with the low pass filter (LPF). To see this consider that when the inputs are offset in phase by 90 degrees, they are orthogonal and their product will have a second harmonic but zero mean, with the result that the output of the LPF shall also be zero. When the frequency of the input signal changes, the phase difference will change and the product will no longer have a zero mean. The output of the LPF will be equal to the new non-zero mean, which is proportional to the phase offset less 90 degrees.

The loop gain will amplify the LPF output signal at the input to the VCO, changing the frequency of the VCO output so that a new equilibrium is reached. At this new equilibrium the input and VCO frequencies are equal, but because they are not equal to the center frequency, the phase offset is no longer 90 degrees and the LPF output is no longer zero. This non-zero output of the LPF is the demodulated signal, since it is proportional to the offset of the instantaneous frequency from the center frequency.

FM discriminators encounter problems in the presence of noise. Firstly, given additive

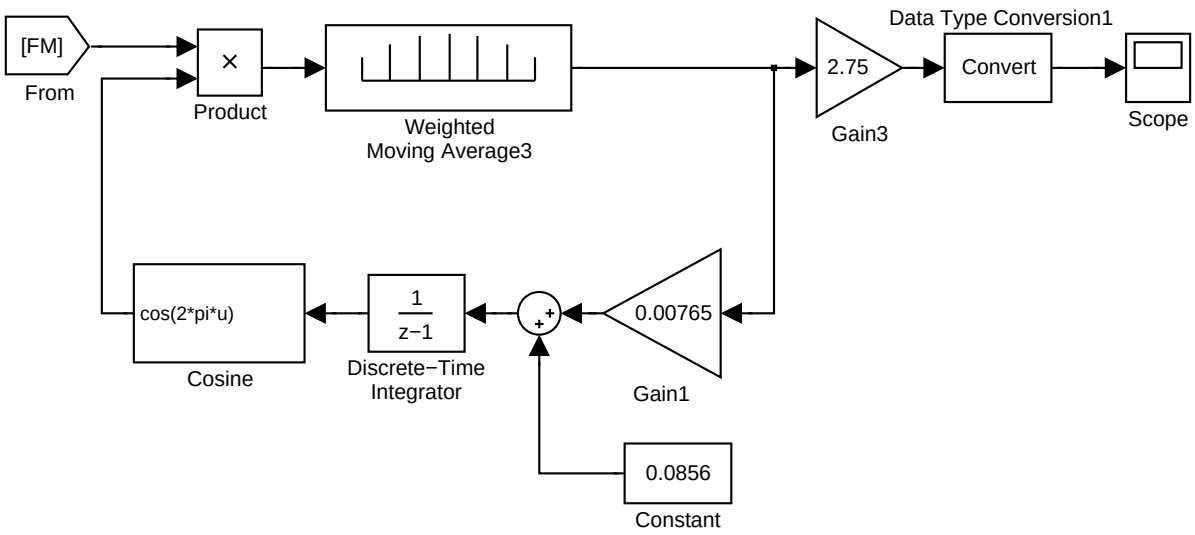


Figure 3.5: *Simulink model for a digital FM demodulator.*

white noise at the input, the noise power at the output increases with the square of frequency [48, p. 305]. Secondly, the so-called *threshold effect*, where spikes appear on the output, occurs when the signal-to-noise ratio (SNR) decreases below some theshold [48, pp. 309-311]. These effects must be taken into consideration when designing a receiver.

With this in mind, we note that the phase-lock-loop is superior to the differentiating discriminator for two reasons:

1. The PLL can be shown to provide a maximum likelihood estimation of phase [48, pp. 502-504], and is therefore the optimum receiver for FM signals, where the modulating signal is encoded in the phase of the carrier.

2. The PLL reduces the SNR at which the threshold effect occurs [48, p. 317]

In the same way that FM modulation can be performed in the digital domain, so can demodulation. A digital PLL (DPLL) uses an NCO instead of a VCO, as is shown in the Simulink model of Figure 3.5.

In order to modulate or demodulate an FM signal, certain parameters are required. These include amplitude, center frequency and frequency deviation. In order to evaluate our system, we must assign values for these parameters so that we can test whether our algorithms produce the correct outputs in response to their inputs.

While developing the concepts of specification, implementation and transformation, we used the DPLL algorithm as described above with the aim of demodulating an FM signal. The parameters of the various components in the DPLL were set for the reception of an FM signal with a center frequency of 10 kHz and a frequency deviation of 5 kHz. The message

signal used to modulate the signal was a short chirp from 10 to 500 Hz. These parameters were chosen to make the signal easy to visualise, as shown in Figure 3.6, and fast to simulate. In Chapter 7 we use a more practical FM signal, with a significantly higher center frequency, in order to test the full working system.

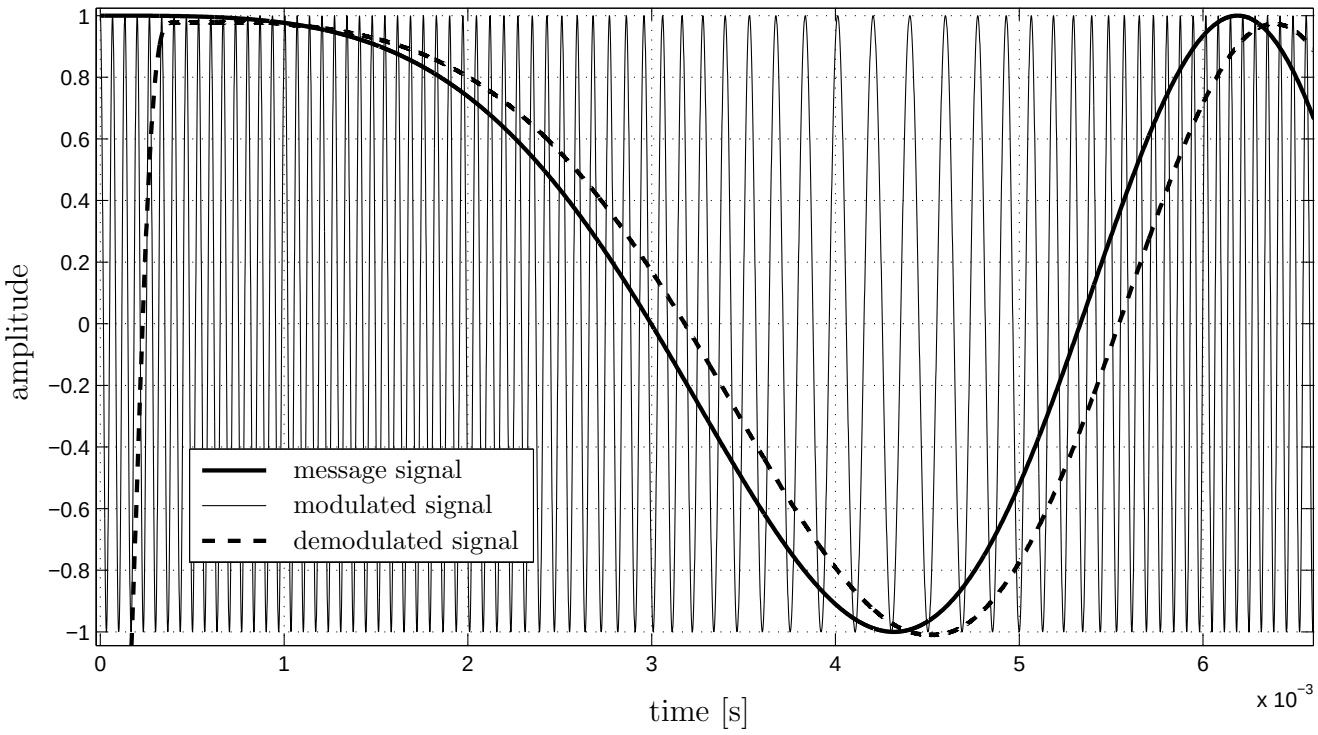


Figure 3.6: *Simple FM signal.*

3.4 Reference platform

A platform is required in order to evaluate the suitability of FPGA devices for SDR, and to test whether the framework developed in the subsequent chapters performs correctly. This platform must provide sufficient logic resources as well as data conversion facilities. The Cyclone II DSP Kit [8], pictured in Figure 3.7 from FPGA vendor Altera was selected, and offers the following features:

- 12 bit DAC capable of 165 MSps,
- 14 bit ADC capable of 125 MSps,
- 32 bit audio ADC capable of 96 kSps,
- 32 bit audio DAC capapble of 96 kSps,

- SRAM-based configuration, and
- integrated configuration controller and non-volatile memory.

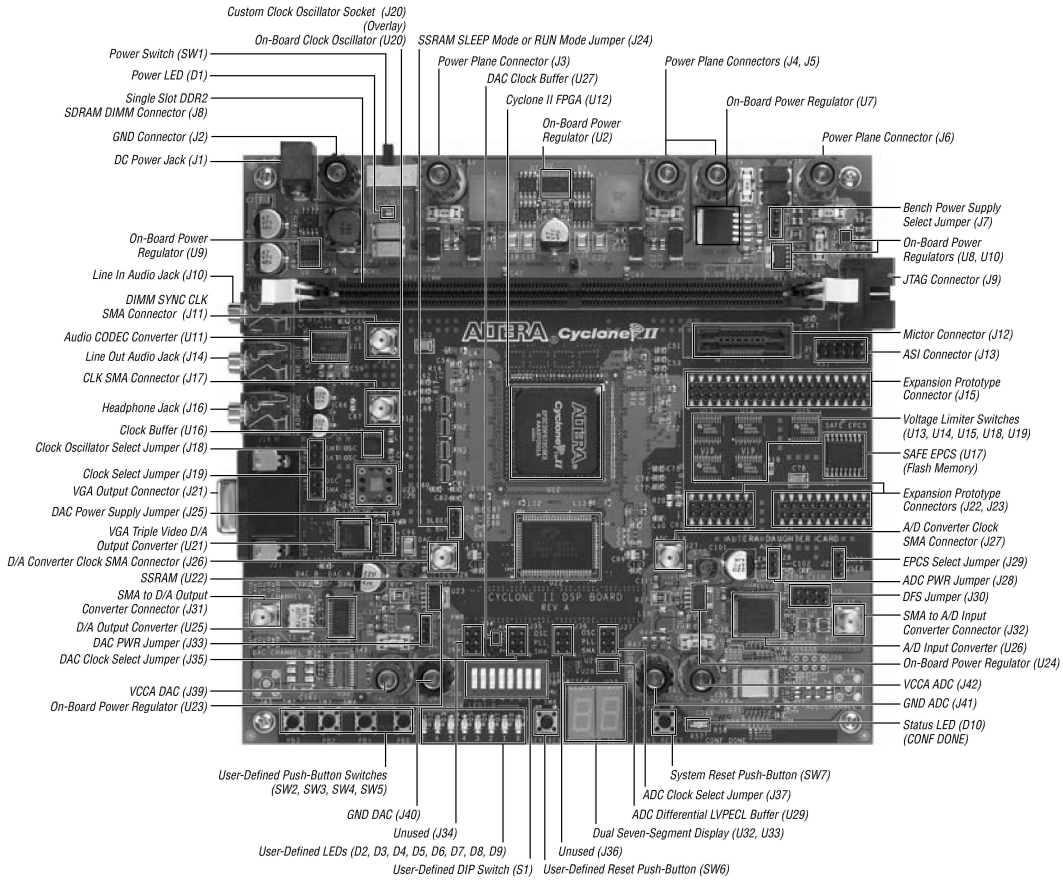


Figure 3.7: *Altera Cyclone II EP2C35 DSP development board.*

Chapter 4

Waveform specification

The selection of an appropriate convention for describing waveforms is central to the goal of portability. The simplest approach would be the use of normal mathematical equations such as Equation 3.1 to describe the relationship between the message signal and the modulated signal. However, not all functions are invertible, such as the cosine operation in Equation 3.1 which prevents us from writing $x(t)$ in terms of $y(t)$ to obtain a formula for demodulation.

For this reason, it is common in the field of Telecommunications to describe a signal in terms of the systems used to generate or process it. This distinction between *signals* and *systems* is important. By specifying a waveform in terms of it's modulation and demodulation algorithms, known as constructive modelling [16], ambiguity is eliminated. This is also useful when several algorithms exist, since the most obvious of these may not be the most efficient or the most robust in the presence of noise.

We shall therefore focus on defining waveforms in terms of the algorithms best suited to their modulation and demodulation. This can be done using mathematical equations, block diagrams, or both. The suitability of block diagrams for such specifications lies in their ability to describe concurrency, a topic which shall be explored below.

It is also important to keep in mind that we are interested in digital algorithms for waveform modulation and demodulation. We therefore explain the importance of models of computation, and discuss the need for hierarchy and extensibility in specifications.

With the above criteria in mind, we evaluate how specifications can be formalised, and conclude that a domain specific language (DSL), based on a simple XML schema, is appropriate for the task.

4.1 Dataflow

Dataflow is a method for describing computation in terms of the flow of information, and is is a departure from the sequential style of computer programming, where structure reflects the ordering of operations.

While sequential programs model data by using variables, and computation by using

procedures, dataflow programs are specified in terms of a graph, with nodes (or vertices) representing computations and arcs (or edges) representing data. We note that sequential programs can also be represented in graphical forms, such as *statecharts* [27], but in this case nodes represent states and arcs represent state transitions, a model which is distinct from dataflow.

Dataflow is especially useful for DSP applications, as is demonstrated by the widespread use of block diagrams to describe DSP algorithms.

In the SU SDR system, we refer to the dataflow nodes as *converters* and the arcs as *links*. In order to fully describe the behaviour of a modulation or demodulation algorithm (such the the DPLL introduced in Chapter 3) using dataflow, we must specify both the interactions between converters (inter-converter specification) and their internal behaviour (intra-converter specification).

4.2 Inter-converter specification

Dataflow graphs can be interpreted in many ways, known as *models of computation*. Models of computation assign semantic interpretations to the graphical syntax of the graph. We shall restrict our interpretation to a model of computation known as *synchronous dataflow* (SDF) [34] and depicted in Figure 4.1.

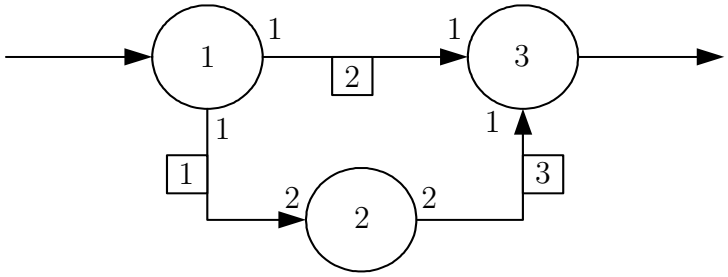


Figure 4.1: *Synchronous dataflow (SDF) graph.*

In SDF, data is transferred in discrete units called *tokens*, and along any given arc, these tokens may travel in only one direction. Arcs perform the function of first-in first-out (FIFO) queues. The nodes in the graph are known as *actors*, and process input tokens to generate output tokens. This can only be done when sufficient tokens are available on all input arcs. The number of tokens which must be available on each input arc is known as its input port rate, and is fixed. When this number of tokens is available on each respective input arc, the actor *fires* (executes), and produces a fixed number of tokens on each output arc. The number of tokens produced on each output arc is known as the output port rate. An SDF graph with port rates is shown in Figure 4.1. An arc may hold initial tokens, which

effectively implement delays, and the formal notation for this is shown in figure 4.2.

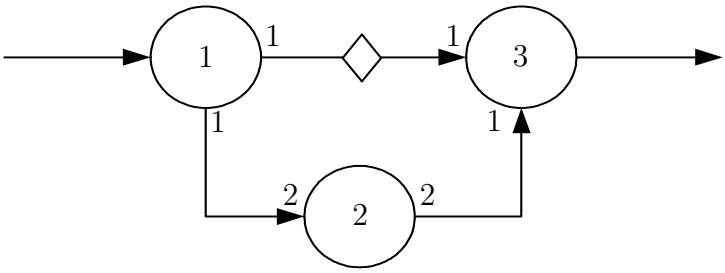


Figure 4.2: *SDF graph with initial token as a delay element.*

SDF graphs are well known for their usefulness in describing DSP applications. Apart from their ability to model both signals and systems, they have additional advantages in terms of formal analysis, namely that they make deadlock and memory boundedness decidable properties [34]. Figure 4.3 shows an SDF graph for the reference design DPLL, which would result in deadlock were it not for the delay in the feedback path. In this figure, the conventional unit delay symbol is used to denote the initial token. The superscript denotes the number of delays, so z^{-1} indicates unit delay while z^{-2} indicates a delay of two samples (or tokens). The absence of a port rate is interpreted as a default port rate of one. SDF graphs in which all ports rates are unity are very common and are referred to as being *homogeneous*.

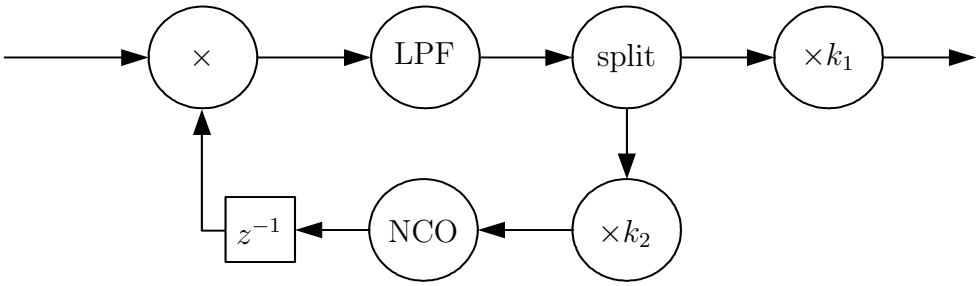


Figure 4.3: *SDF for the DPLL with initial token permitting feedback.*

SDF also has the useful feature that it abstracts the notion of time so that we do not have to assign a duration (sampling frequency) to the interval between tokens. We can therefore describe our systems using units such as cycles per sample, instead of cycles per second (hertz), an important distinction in digital signal processing.

In the nomenclature used for the SU SDR project, SDF actors or nodes are analogous to SDR converters, and SDF arcs (or edges) are analogous to SDR links. Later in this chapter we shall show how an SDF graph can be fully described in a text-based specification.

4.3 Intra-converter specification

The previous section deals with the interaction between actors in the system. We turn now to the internal definition of actor behaviour, which is specified by describing how input tokens are mapped to output tokens. This task is similar to writing a function in most common programming languages, such as C++. A sequence of statements specifies how the inputs should be manipulated (using expressions), and what values should be written to the outputs (using assignments).

Consider an actor whose task is to function as a gain block and, upon firing, multiplies the sample at its input port by a constant and writes the result to its output port. The C++ in Listing 4.1 is sufficient for expressing this behaviour if the names used allow input and output ports to be identified. No reference is made in this statement to data types or other implementation details, which is desirable when specifying behaviours which are intended to be portable.

Listing 4.1: *C++ code for the internal behaviour of a gain actor.*

```
gain_output = gain_input * gain_value;
```

Another feature which is required for specifying many behaviours is the concept of memory between firings. It should be possible for an actor to store some information in one firing and retrieve it in the next. For example, an accumulator (or integrator) must add its current input to its previous output to obtain its current output. Once again if names can be assigned to such storage elements, then the behaviour can be specified using simple expressions. For the case of the accumulator, consider Listing 4.2 which specifies the behaviour using C++, under the assumption that the value of the `accumulate` variable is persistent between firings.

Listing 4.2: *C++ code for the internal behaviour of an accumulator actor.*

```
gain_output = gain_input + accumulate;
accumulate  = gain_output;
```

Having determined methods for specifying both inter- and intra-actor behaviour, we now look at domain specific languages (DSLs) appropriate for codifying these behaviours with a concrete syntax.

4.4 Waveform Description Language (WDL)

WDL is a DSL, proposed by Willink [47], specifically designed for the specification of modulation and demodulation algorithms. The language has a concrete textual syntax, and semantics which combine several models of computation. These include:

- the *token flow* model, a superset of SDF,
- the *continuous time* model, which is used to represent analogue signals, and
- the *finite state machine* model, used for algorithms with various modes.

The proposal also places an emphasis on hierarchy using block diagrams. A graphical syntax is proposed where each block, or *entity*, can be either a composite entity (a combination of other blocks), or a leaf entity.

As of 2006, the language specification is incomplete and is not being actively developed. However, the principles described in the available publication [47] have influenced the design, described below, of a simple DSL using XML for the specification of waveforms.

4.5 Extensible Markup Language (XML)

XML [45] is a general purpose markup language standardised by the World Wide Web Consortium (W3C) and popularised by its use on the internet. It is a textual syntax for representing data in a manner which is both machine and human readable. The representation of information is hierarchical and has a tree structure. Elements are delimited with start and end tags, and may contain attributes, text, or other elements. Examples of XML can be seen in Listings 4.3 and 4.4.

In this section we describe an XML schema for specifying modulation and demodulation algorithms. A schema is a set of rules stating the names of elements and attributes which may be used and what their relationships to each may be in terms of the hierarchy.

The goal of the schema is to allow XML to be used to describe a dataflow graph. As was previously mentioned, nodes in such a graph are called *converters* and arcs are called *links*. Each converter has an associated *process* which describes the intra-converter behaviour as discussed in Section 4.3, and may have *attributes* which can implement the memory requirement also described in that section. Attributes can also be used to parameterise blocks, and the C++ implementation platforms allows these to be adjusted during execution. This is not the case with the VHDL platform, where attributes can only be set at compile time. These implementations will described fully in Chapter 5.

All of the above-mentioned concepts are provided for in the XML schema as elements of the same name, with the exception of nested converters, which are instantiated using

the `component` element. Listing 4.3 shows the top level XML specification for the DPLL introduced in the previous chapter.

Note that the complete DPLL is itself seen as a converter, with inputs and outputs. Inside all converters, behaviour is specified using the `process` element. In this case the behaviour is specified by instantiating further actors and links between them. This is known as a composite actor, hence the attribute `syntax="composite"`.

Leaf converters, on the other hand, do not contain other actors. Instead, their behaviour is described using a code stub as described in the Section 4.3. An example of such an actor specification is provided in Listing 4.4. Multiple `process` elements are allowed if they contain code stubs of different languages, denoted by the `syntax` attribute, and none of them specify a composite behaviour. These converter specifications can reside in separate files stored in a library.

While XML is both machine- and human-readable, it is not particularly easy to edit by hand. However, its structured nature allows the use of editing environments which improve usability. These are readily available due to the widespread use of XML. Such editors may automatically close elements according to some rule, or could hide the element syntax altogether and present the information visually.

4.6 Conclusion

In this chapter we have formalised the way in which software algorithms for waveform modulation and demodulation are specified. The XML schema described was already partially in place as part of the SU SDR project, but was only used to describe intra-converter behaviour. The extension of the schema to describe composite converters is an original contribution of this work, as is the introduction of SDF to formalise such composite specification.

Listing 4.3: *Top level XML file specifying dataflow graph which implements a DPLL for FM demodulation (some link elements have been omitted).*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<converter>
  <name>sdr_adfm_fmrx</name>
  <author>R Brady</author>
  <date>2006-05-03</date>
  <description>SDR all-digital FM demodulator.</description>
  <summary>This converter implements a digital
           phase-lock loop for FM demodulation.
  </summary>
  <input>
    <name>input</name>
    <description>The FM modulated input.</description>
  </input>
  <output>
    <name>output</name>
    <description>The demodulated baseband audio output.</description>
  </output>
  <process syntax="composite">
    <component name="phase_comp"  type="sdr_product"     />
    <component name="loop_filter" type="sdr_fir"         />
    <component name="loop_gain"   type="sdr_gain"         >
      <set name="gain_value">0.00765*1.1</set>
    </component>
    <component name="bias"        type="sdr_bias"        />
    <component name="accumulator" type="sdr_accumulator" />
    <component name="cos_lut"     type="sdr_cos"         />
    <component name="output_gain" type="sdr_gain"        />
    <component name="out_lpf"     type="sdr_fir"         />
    <link from="phase_comp"
          from_port="output"
          to="loop_filter"
          to_port="input"
          signed="yes"
          length="16"
          frac="14" />
      .
      .
      .
  </process>
</converter>
```

Listing 4.4: *XML specification for the gain actor.*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<converter>
  <name>sdr_gain</name>
  <description>Simple gain block.</description>
  <constant signed="yes" length="16" frac="10">
    <name>gain_value</name>
    <default>1.0</default>
  </constant>
  <input>
    <name>gain_input</name>
    <default>0.0</default>
  </input>
  <output>
    <name>gain_output</name>
  </output>
  <process syntax="VHDL">
    <code>
      gain_output := resize(gain_input * gain_value, gain_output);
    </code>
  </process>
  <process syntax="C">
    <code>
      gain_output = gain_input * gain_value;
    </code>
  </process>
</converter>
```

Chapter 5

Target platforms for waveform implementation

In the previous chapter we developed a useful system for specifying modulation and demodulation algorithms. We must now look for appropriate frameworks on our target platforms, namely microprocessors and FPGAs. These frameworks shall be specified in the C++ and VHDL languages respectively.

C++ and VHDL are high-level languages, each with a powerful syntax and semantics, as well as both standard libraries and additional reusable software libraries. It is important to exploit the language features to whatever extent is possible and practical. Where a language provides a type system for example, it is desirable for our compiler to use that type system instead of synthesising its own. The set of language features and libraries which we select to use in our system forms a target framework. It is important to understand that the target is not only the device or only the language, but a structure specified in that language for that device. The usefulness of this approach is demonstrated by the software framework which was developed by Cronjé [20].

5.1 Scheduling

Scheduling is central to the correct and efficient implementation of a specification. For each actor in the specification, a firing results in the consumption of tokens (samples) at the input arcs and the generation of tokens at the output arcs. Scheduling can be described as the task of deciding how often, and in which order, if any, to invoke the firing of actors. This problem is closely related to the models of computation described in Chapter 4, and several solutions will be explored below.

A scheduling solution must result in semantically correct execution of the specification on the target architecture with minimal (or at least finite) memory and minimal (or at least finite) execution cycles. In this section we explore the solutions available for both target architectures of interest.

5.1.1 Scheduling for μP targets

In μP platforms we are usually restricted to a single processing element, or CPU. Platforms are now available with two or more processing elements, but the programming model remains sequential. For simplicity, we assume a single processor in the discussion below.

This single shared processing resource means that only a single converter in the system can be executing at any given time. However, all of the converters must process their data regularly so that data continues to flow through the system as required. This introduces a need for time sharing of the processing unit.

One solution involves the use of large FIFO buffers (or small buffers which can grow) to implement the arcs of the SDF graph. Procedures (respresenting the computation of nodes) are invoked repetitively regardless of the availability of tokens, but only actually fire (generate output tokens) when a token becomes available. There are at least two possible ways to achieve this:

1. launch each actor in its own thread of execution and make read calls to the FIFO blocking, or

2. use a single thread of execution in which actors are invoked sequentially (according to a round robin schedule) and conditionally fire or do not fire depending on the availability of tokens, but do not block.

These two options are equivalent and implement a model of computation known as *dataflow process networks* [35] which is a superset, or extension, of SDF. Bounded memory execution is not guaranteed by the above model, but may be ensured by extending it so that firings occur on condition that none of the FIFOs at an actor's outputs are full to capacity. This capacity must be specified for each FIFO, or may default to a certain number of tokens.

Another solution to the scheduling problem uses the SDF specification to perform compile-time analysis and determine:

- whether or not the SDF graph will deadlock,

- whether or not the SDF graph may be executed in bounded memory,

- what schedule of invocations will result in bounded memory successful firings,

- what capacity should be assigned to each FIFO buffer.

This technique is known as static scheduling, and is described by Lee and Messerschmitt [36] for single- and multi-processor systems, but futher analysis is beyond the scope of this work.

5.1.2 C++ framework

The dataflow process network approach described above forms the premise on which the SU SDR project's *libsdr* software operates. The software is implemented in C++ and the structure is shown in Figure 5.1. Object-oriented design is used extensively.

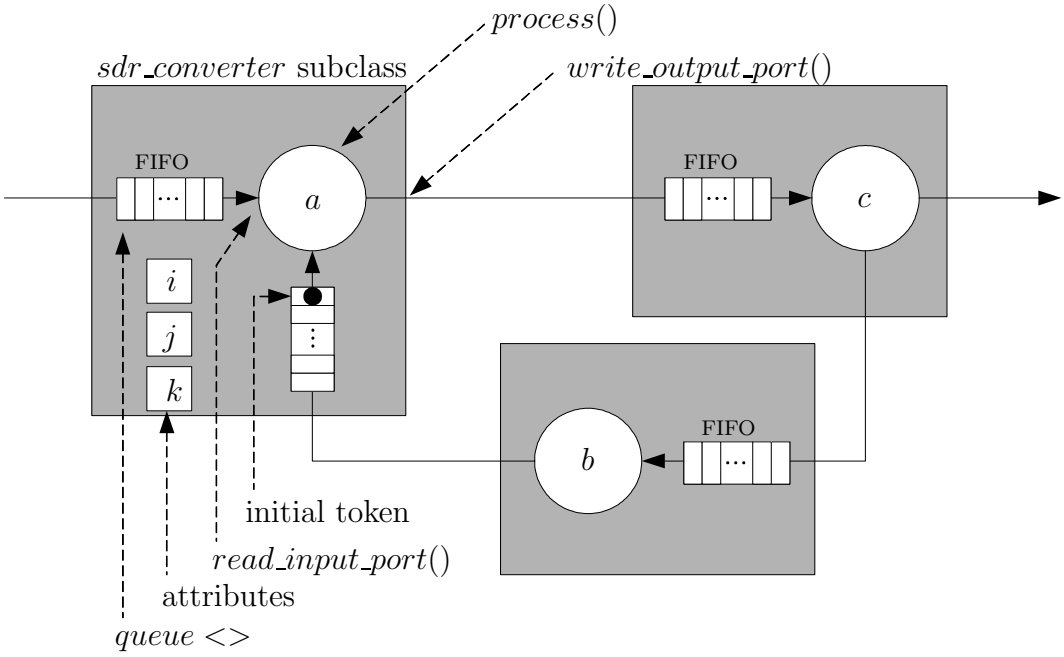


Figure 5.1: *Informal diagram describing C++ software structure of a libsdr radio.*

The converter primitives are defined as a subclasses of the *sdr_converter* base class. This inheritance ensures that each converter conforms to a certain structure which ensures it can interact with other converters once instantiated. Links are implemented as FIFO buffers at the converter inputs where they terminate, using the C++ `queue` container. A converter has a pointer for each of its outputs, which points to the FIFO queue for the converter and input which are linked to that output. Each converter must implement a `process()` method. This method reads samples from the inputs by removing elements from the front of the FIFO queues, and writes samples to the outputs by dereferencing the above-mentioned pointers and inserting elements into the resulting FIFO queue. Attributes are stored as member variables of the converter class.

These classes are instantiated by a top-level program which is also responsible for linking the converters together. This is not done by the manual assignment of pointer values but rather using a `link()` function which takes the names of endpoints as arguments. This top-level program is also responsible for executing the graph, usually by entering a loop in which the `process()` functions of the instantiated converters are called according to some schedule. This could be a simple round-robin schedule, but the modular nature of *libsdr* also

makes it possible to implement other techniques such as the static scheduling mentioned in Section 5.1.2. Reseach on this technique is underway at the SU SDR project [24].

5.1.3 Scheduling for FPGA targets

Field programmable gate arrays differ significantly from μP platforms in that they permit many computations to occur concurrently and independently of each other. The scheduling task is to determine how many individual computational units to instantiate, what computation should be performed by each, and how to map them to the actors and firings of the SDF graph (converters and processes of the *libsdr* system). Several options are available, and we shall explore these with a view to obtaining a concrete solution which can be codified in VHDL.

It is appropriate at this time to discuss vendor neutrality with respect to VHDL. VHDL code which compiles successfully using the synthesis software of one vendor, does not always do so with the software of others. This has two causes. Firstly, interpretations of the VHDL standard may differ between vendors. As an example, Altera Quartus forbids a *process* construct to be labelled with a name already used for a signal or port, while Synplify Pro does not impose this restriction. The second cause of incompatibility is the vast number of vendor-supplied libraries and tools which are not portable due to licensing or technical restrictions.

We therefore make it our goal, in the development of VHDL for SDR, to avoid vendor supplied libraries and to have all VHDL code compile successfully with at least two synthesis tools, namely Altera Quartus and Synplify Pro. Quartus was selected because the devices used for evaluation (Cyclone II and Stratix families) are made by Altera, and Synplify Pro because it can re-generate code for almost all other vendors.

We return now to our goal of implementing SDF on the FPGA by considering three candidate solutions.

Soft processor with hardware acceleration

Soft processors are microprocessor designs which may be programmed onto FPGAs. They conform to either the Von Neumann or Harvard architecture and therefore require memory for data and instructions. Examples include the Nios II from Altera [9] and those designs which are available as both hardware implementations and soft processor specifications, such as the AVR from Atmel [10].

An SDR system would be implemented by writing sequential code to run on the embedded processor. Soft processors are made available with compilers for common languages, such as C++, which can be run directly on the processor or on top of a minimal operating system. This would allow *libsdr* (possibly with modifications) to execute on this platform

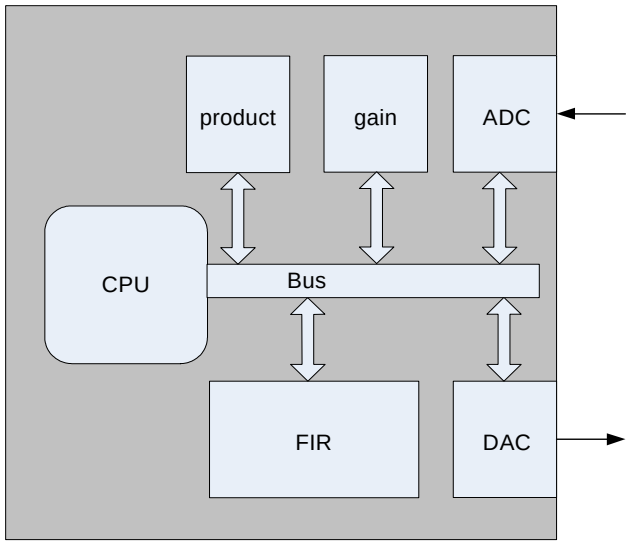


Figure 5.2: *Softprocessor configuration with hardware acceleration.*

implementing the SDF specification. The advantage of using an FPGA here is that expensive operations can be executed by specialised hardware.

This is done by having a system bus to which the processor and several custom hardware accelerators are connected. These accelerators perform repetitive or expensive tasks, and could correspond to fine-grained operations such as multiplication, or to large-grained operations such as FIR filtering. This topology is shown in Figure 5.2.

The processor is then scheduled as described in Section 5.1.1, but is additionally responsible for the dispatch and collection of input and output data from each acceleration unit.

We reject this solution for the development of our SDR software because:

- it applies a sequential programming model to a parallel device, preventing us from fully evaluating its parallel processing ability,
- it reduces the scheduling problem to a speed optimisation of the work already performed,
- it requires all data to pass through the processor and shared bus at some point, creating a bottleneck which prevents maximum exploitation of the throughput capability of the FPGA, and
- there is a lack of soft-processors available with the required combination of sufficient performance and vendor neutrality.

Parallel execution

Significant research [29, 6, 46] has been performed to determine effective techniques for mapping SDF and other dataflow models directly onto parallel hardware without a soft processor. A concise summary is provided by Jung and Ha [32]. The main trade-off explored by different approaches is that of area versus throughput.

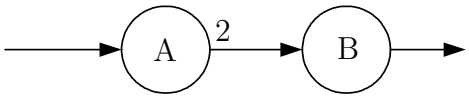


Figure 5.3: *SDF graph with upsampler and one other operation.*

For maximum throughput, the Ptolemy [46] system creates a VHDL entity for every firing of each actor in a periodic schedule. To understand this, consider the SDF graph in Figure 5.3. For each firing of actor A, actor B must fire twice. This can be repeated continuously and is therefore known as a periodic schedule. Ptolemy will generate one VHDL entity for actor A and two for B, as shown in Figure 5.4. Because actors may have memory, state information must be passed between the entities which represent consecutive firings. The advantage of this system is that control logic is not required since a complete execution of the entire SDF graph occurs synchronously in each clock cycle.

A more intuitive approach is to generate one entity for each actor in the system. Because actor A must only fire once for every two firings of actor B, control logic is required. Two methods are used for this. The first, described by Meyr *et al.* [29], uses centralised control. A counter based state machine must control the firings of each actor according to some schedule, which may be statically determined at compile time. This arrangement is shown in figure 5.5.

The system proposed by Adé *et al.* [6] also maps single actors to single logic entities, but uses a handshaking protocol between these entities instead of a centralised controller, as

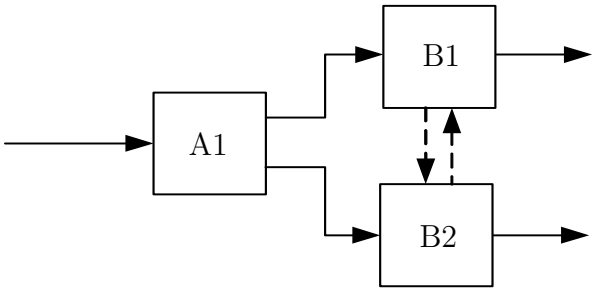


Figure 5.4: *One entity per firing.*

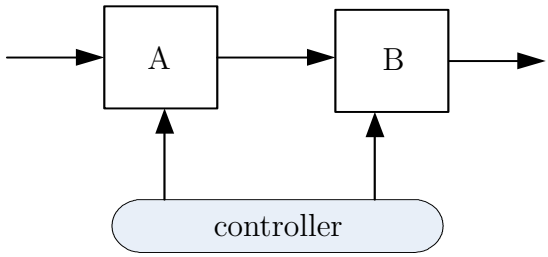


Figure 5.5: *One entity per actor with centralised control.*

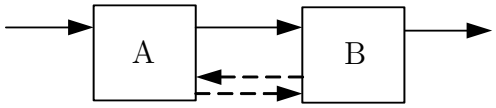


Figure 5.6: *One entity per actor with a handshaking protocol for distributed control.*

shown in Figure 5.6. Output ports are accompanied by handshaking signals which indicate the availability of data, and input ports by signals indicating their readiness to accept data. These signals incur non-negligible overhead [32] but the method does offer a compromise between the high area requirements of the Ptolemy [46] system and the complex control required for the system proposed by Meyr *et al.* [29].

5.1.4 VHDL framework

For our proof of concept we aim to demonstate working VHDL code for the execution of homogenous SDF graphs. For this case, one sample is transferred over each link or processed by each converter during each clock cycle, and handshaking signals are not required.

Each converter in the specification is mapped onto a single VHDL *process* within the architecture body, and each link becomes a *signal* declaration. The process statements are sensitive only to the rising edge of the global clock, making the entire design synchronous.

Attributes from the specification are implemented as process variables. Process variables are distinct from signals in that:

- they have limited scope (they are only visible inside the process statement),
- they take on assigned values immediately (using the `:=` operator) and not at the end of the process statement (which is the case with signals which use the `<=` operator), and
- they may be assigned more than once inside a process statement.

When an attribute's value must be remembered between firings, we need to synthesise a flip-flop register as memory for that data. We know the value must be remembered if the

attribute is used before it is assigned in the code stub. VHDL synthesis tools use exactly this observation to determine whether or not a register must be synthesised for a process variable. Therefore, by writing our code stubs as VHDL which can be included inside a process, we can assign the responsibility of determining whether or not registers are required to the synthesis tool.

To understand this, consider the VHDL process statement in Listing 5.1 which implements a counter. A register will be synthesised for the variable `k` , since it is used on the right hand side of an assignment statement before it is used on the left. This is not the case for `j` and a register shall therefore not be synthesised for this variable.

Listing 5.1: *VHDL process statement with variables.*

```
process (clk)
    variable j,k : integer;
begin
    if rising_edge(clk) then
        j := k + 1;
        k := j;
        output <= j;
    end if;
end process;
```

An example of working VHDL code for an SDR converter can be seen in Listing 5.2. This VHDL is autogenerated, a process which will be described fully in Chapter 6. This is the reason for the `SDR_AUTOGEN` prefix on some identifiers. Suffice it to say at this point that these identifiers represent signals corresponding to the links in the specification.

We have described how process variables can be used to represent attributes from our converter specification. Constants are also declared inside the process, and variables are declared for the input and output ports of the actors. This is an important feature. While it would be possible to assign directly to, and read directly from the signals which represent links in the system, we instead place variables in between these signals and the code stubs which use them. This can be observed in Listing 5.2 where variables with the names `gain_input` and `gain_output` are declared. Ignoring the reset code, the first statement in the process body reads the value of the `SDR_AUTOGEN_loop_filter_output` signal (representing a link) into the `gain_input` variable. The last statement in the process body writes the value of the `gain_output` variable to the `SDR_AUTOGEN_loop_gain_output` signal. This has two advantages:

- All identifiers, including inputs and outputs, can be treated uniformly as variables in the process code. It would be confusing for someone writing a code stub to use the variable assignment operator, `:=`, everywhere except when assigning to the output, in which case they would need to use the signal assignment operator, `<=`.

- Inside the code stub for a converter, references can be made to ports of the converter, not to the links which are connected to those ports and have different names depending on how the converter is used. This allows code stubs to be pasted verbatim from the converter specification into the correct position inside the process body, as shown in Listing 5.2.

Listing 5.2: *VHDL process statement for the gain actor specified in Listing 4.4.*

```
-- Main process with actor firing code
loop_gain : PROCESS  (clk)
    VARIABLE gain_input  : sfixed(1 downto -14) := to_sfixed(0.0, 1, -14);
    VARIABLE gain_output : sfixed(-3 downto -18) := to_sfixed(0.0, -3, -18);
    CONSTANT gain_value  : sfixed(-6 downto -17) := to_sfixed(0.00765, -6, -17);
BEGIN
  if rising_edge(clk) then
  if reset = '1' then
    gain_output := to_sfixed(0.0, -3, -18);
    SDR_AUTOGEN_loop_gain_output <= gain_output;
  else
    gain_input := SDR_AUTOGEN_loop_filter_output;
    gain_output := to_sfixed(0.0, -3, -18);
    -- Start actor code paste

    gain_output := resize(gain_input * gain_value, gain_output);

     -- end actor code paste
    SDR_AUTOGEN_loop_gain_output <= gain_output;
  end if;
  end if;
END PROCESS;
```

Note that flip-flop registers will not be synthesised for the `gain_input` and `gain_output` variables, since they are both assigned before being used. The use of such variables therefore incurs zero area overhead. Also of interest is the synchronous reset code which can be seen in Listing 5.2. When the reset signal is high, all processes assign default values to their variables. this is especially important at power-up when the internal state of registers is not always known.

Because the design is synchronous however (all processes are sensitive only to the rising edge of the clock), registers are synthesised at the output of each converter. This is desirable because it breaks long combinatorial paths which could pass through several converters, severely reducing the achievable clock speeds. Figure 5.7 shows the structure of a synthesised converter.

In this section we have demonstrated working VHDL for the implementation of SDR converters. The internal behaviour of each converter in the system is represented using a VHDL process. The interaction between converters, specified as links in the dataflow

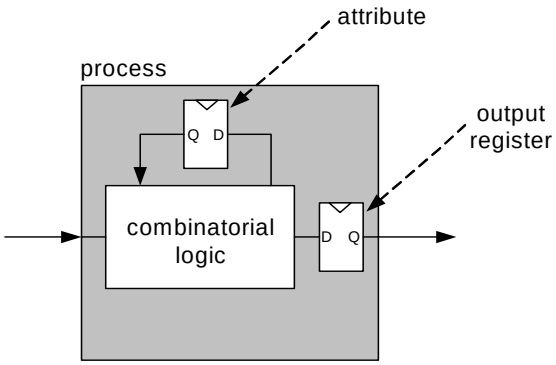


Figure 5.7: *Synthesised structure corresponding to a VHDL process which implements an actor.*

representation, is achieved by simply declaring VHDL signals for each link and making the individual converter responsible for reading from, and writing to them, as shown above.

5.2 Data types

In digital signal processing, signals are represented as discrete valued samples located at discrete locations in time or frequency. Discreteness in time is handled by our SDF model, but we also require a means to represent discreteness in value. Solutions to this problem are known as *data types*.

There are two important considerations when reviewing various data types. First, a data type must be able to represent a number with finite storage while incurring minimal loss of precision. Secondly, the data type must allow mathematical operations to be performed within resource constraints. Resources may include time, power and logic gates.

Integer numbers are generally simple to store and manipulate. Real numbers, used extensively in digital signal processing, present more of a challenge. A popular solution is the floating-point representation and its accompanying arithmetic operations. This technique is currently used in the SU SDR implementation for C++ platforms. Many C++ targets (usually GPPs) contain a floating point unit (FPU), an arithmetic co-processor specifically designed to perform mathematical operations on such numbers.

Such an approach is not appropriate for FPGA targets due to area constraints, since floating-point arithmetic requires unrealistic amounts of logic [40]. The method most readily used for DSP on FPGAs is fixed-point representation and its accompanying arithmetic.

Fixed-point numbers are represented in much the same way as integers, using a certain number of bits to represent increasing powers of the radix, which is usually 2. In addition, a radix point is specified to exist between some pair of bits within this sequence. Bits to the left of this point represent positive powers of the radix, starting at zero and increasing with

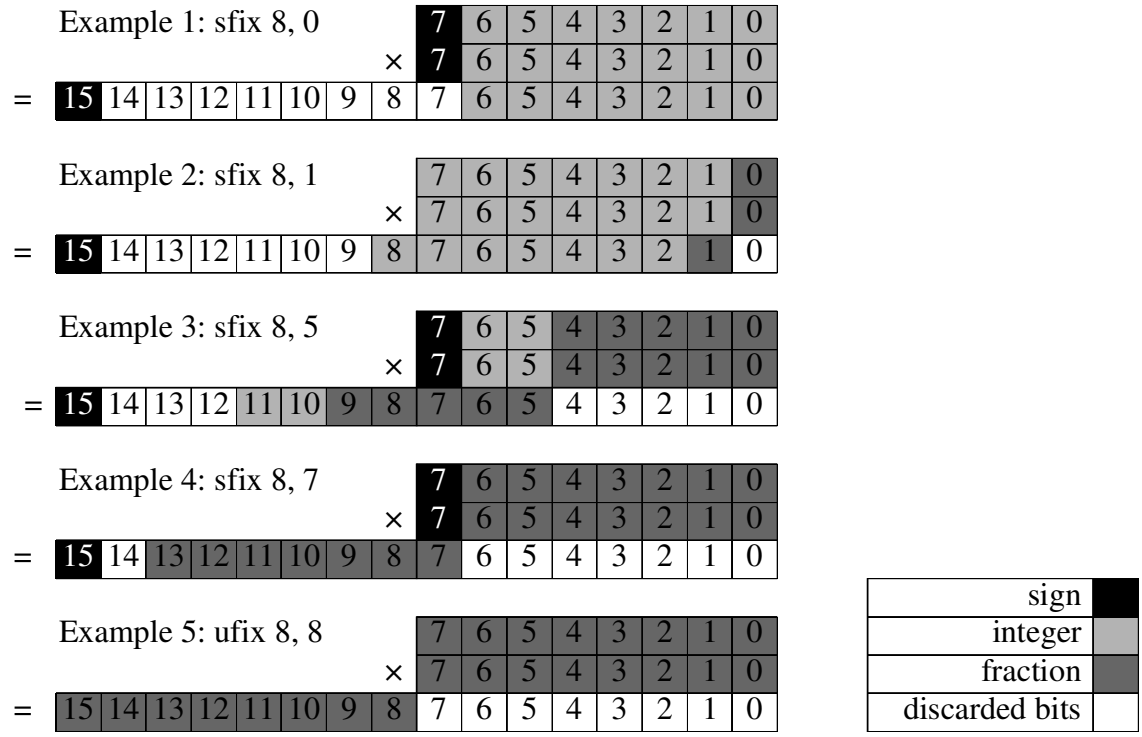


Figure 5.8: *Fixed point data type for real numbers, with radix 2.*

each bit. To the right of the radix point, bits represent increasingly negative power of the radix.

Addition under this scheme occurs in much the same way as addition for integer data types. Two's complement can also be used for subtraction and the representation of negative numbers, as with integer types. Multiplication is more complex, as it required an additional step after performing integer style multiplication. This step involves shifting the radix point appropriately. If the same bit length is to be used for the operands and the result, overflow becomes possible.

Figure 5.8 provides several examples of multiplication with different positions of the radix point, along with details of how the result must be resized to retain the operand length. As shown, certain bits must be discarded and others shifted.

A fixed-point data type was adopted for the proposed system. However, writing VHDL code for fixed-point operations can be cumbersome. The IEEE library for VHDL provides the `numeric_std` package for manipulating signed and unsigned integer data, and this is used in most systems requiring fixed-point operations [39]. Each multiplication and division must be followed by a bit-slicing operation to discard the unwanted bits shown in Figure 5.8.

This problem was solved by using the `fixed_pkg` [11] package, a draft standard from the IEEE. Like `numeric_std`, it represents numbers using unconstrained arrays of the

`std_logic` type. This allows any numeric data of any bit length. These can be signed or unsigned depending on whether the `sfixed` or `ufixed` type is used. The novelty of this data type however, is its use of negative array indices to denote the location of the radix point. For example, a declaration of type

```
sfixed(2 downto -5)
```

will indicate an 8-bit, signed, fixed-point number with 3 integer bits (including the sign bit) and 5 fractional bits. The arithmetic operations for these types are also defined inside the package and use this additional information to determine and encode the correct radix point in the result.

The package also departs from the convention of `numeric_std` in that the result of arithmetic operations is not the same size as the largest operand. Instead, the result type is just long enough to store the largest (or smallest) possible result of the operation. This is one longer than the longest operand for addition, and is the sum of the operand lengths for multiplication. The programmer then uses a simple `resize` function to obtain the desired bit length. The power of this solution lies in its ability to specify arithmetic operations in a manner which is polymorphic with respect to both bit length and radix position.

This polymorphism is a feature of the `fixed_pkg` package. The functions inside this package responsible for arithmetic operations are able to inspect the radix postions and word lengths of their operands and set these properties correctly for the result.

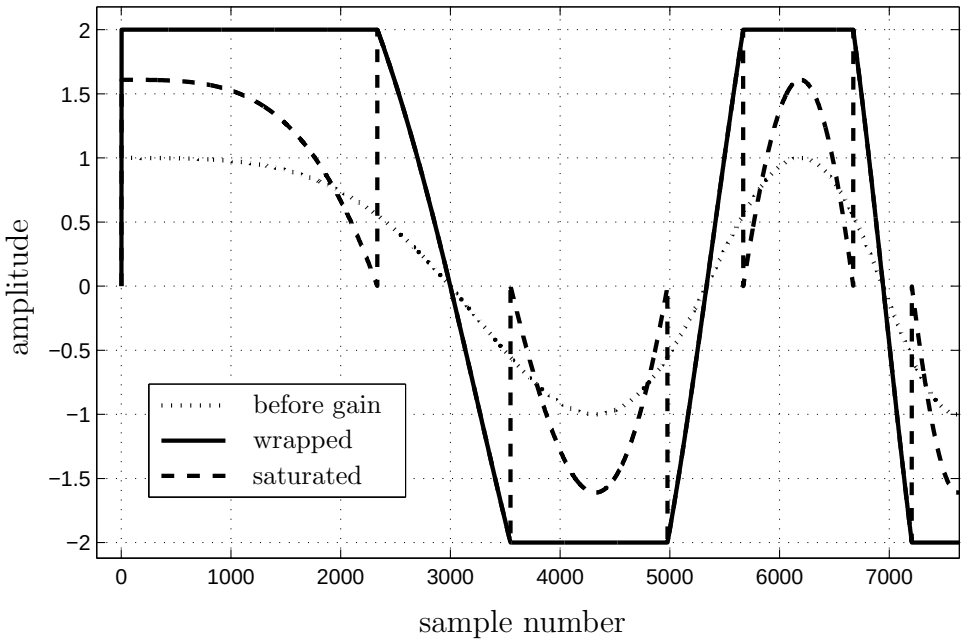


Figure 5.9: *Comparison of saturation and (sign-preserved) wrapping when overflow occurs during a multiplication operation with fixed-point data.*

The `fixed_pkg` package has an additional advantage. When discarding bits, overflow

can occur and this usually results in wrapping. Simply put, any number just too large to represent becomes a very small number (for the unsigned case), or a very large number with the opposite sign. The `resize` function solves this by allowing saturation arithmetic, where numbers too large to represent are simply set to the largest number which can be represented. This important distinction is demonstrated in Figure 5.9 where a signal is amplified and becomes too large. The `numeric_std` package would wrap but preserve the sign, while the `fixed_pkg` package simply saturates the number.

5.3 Conclusion

In this chapter we have given details of solutions to the two major problems when targetting SDR algorithms to FPGA devices. The problem of scheduling was solved using VHDL processes arranged to implement the SDF model of computation, and the problem of real number representation was solved using fixed-point data types based on a draft IEEE standard library.

The structured nature of the code is suitable for autogeneration, as detailed in the following chapter.

Chapter 6

Autogeneration of waveform software

The research described above has led to the design and implementation of a software program for the automatic generation of VHDL code, which will be described in this chapter. The design builds upon previous work towards autogeneration of C++, and is a key contribution of this thesis. Because VHDL is not an object-oriented language, and due to the requirement for variable word length fixed-point data types, the transformation system targetting VHDL is significantly more complex then the existing system which targets C++.

We first assess traditional compilation techniques and their suitability for the task. We then consider some newer developments in the field of autogeneration of code, namely XML for intermediate forms, and XSLT for transformation. We discuss the design considerations and motivate the relevant design choices. Having laid this foundation, we fully describe the compilation process which was designed as a proof of concept.

6.1 Traditional compilation techniques

The compilation of text-based specifications into other forms closer to the machine level is a well researched field [5, 7]. The process begins by converting the textual representation (source) into a form which can be manipulated by the computer. This task is known as parsing and involves the use of a formal grammar. A formal grammar is a set of rules which govern the source so that the structure of the specification is machine-readable. A parser uses these rules to construct an internal data structure known as an abstract syntax tree (AST).

Several programs exist which construct the parser program automatically given a formal grammar [7]. These are known as *compiler-compilers* or *parser generators*. Their input is a grammar, usually specified in Backus-Naur form [7], annotated with executable *action code*. The parser generator generates source code for a compiler in some target language, and inserts the appropriate action code where specified.

The action code is usually responsible for populating an AST. Once parsing is complete, a series of software routines manipulate the AST to produce new data structures (intermediate

forms) and eventually generate target code.

This approach was evaluated using the ANTLR [38] compiler-compiler which produces Java programs. ANTLR was selected for its modern design, and because of the good library support for data structures in Java. However, the technique was found to be inappropriate for the following reasons:

- A formal language as described above must be designed to represent the structure of the specification. Compiler-compilers cannot accept all formal-grammars and therefore impose restrictions on the grammars which they can accept. Care must be taken to ensure that the language design is compatible with the compiler-compiler being used.
- Changes to the specification language are time consuming, as they require modifying the grammar and action code, and rebuilding the compiler.
- The intermediate forms are computer data structures and therefore require a debugger to be viewed. A custom routine could be written to serialise them to text, but this would need to be modified each time the structure of the intermediate form is changed.

6.2 Extensible Stylesheet Language for Transformations

The Extensible Stylesheet Language for Transformations (XSLT) [44] is a language for transforming XML documents into other XML documents. XSLT is itself a subset of XML. An XSLT processor, such as *xsltproc* [3], takes one or more XML documents and one XSLT document as input, and applies the transformation specified in the XSLT document to the contents of the XML documents to produce an output XML document. This is depicted in Figure 6.1.

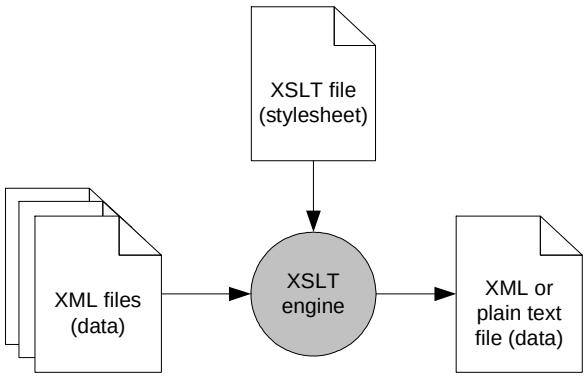


Figure 6.1: *Transformation process with XML and XSLT.*

XSLT has an additional feature which is important to our cause. It has the ability to generate plain text output instead of well-formed XML. This makes it ideal for the generation

of C++ and VHDL code.

The XSLT engine enumerates the input (source) tree and looks for nodes which match the template rules specified in the XSLT file. When a matching node or node set is found, the instructions associated with that template rule are executed on the node set which results in new nodes being created and added to the output (result) tree. This process is known as *template matching* [28].

Listing 6.1: *Example of an XSLT transformation.*

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>
    <xsl:template match="specification[actor]">
        <xsl:for-each select="actor">
            <xsl:value-of select="@name"/>
            <xsl:text>: process (clk) begin end process;&#10;</xsl:text>
        </xsl:for-each>
    </xsl:template>
</xsl:transform>
```

A small example of this process is provided in Listings 6.1 through 6.3 to introduce the concept of template matching and demonstrate the ease with which output text can be generated. In this example the output is plain text rather than well-formed XML, as specified by the `xsl:output` node in Listing 6.1. However, this generated text is valid VHDL, and in this case two empty `process` statements are generated. The `xsl:` prefix in front of element names in Listing 6.1 is a namespace operator used to distinguish XSLT instructions from normal XML.

Listing 6.2: *Example input XML for the XSLT transform of Listing 6.1.*

```
<specification>
  <actor name="actor1"/>
  <actor name="actor2"/>
</specification>
```

Listing 6.3: *Output generated by applying the transform in Listing 6.1 to the input in Listing 6.2.*

```
actor1: process (clk) begin end process;
actor2: process (clk) begin end process;
```

The XML tree in Listing 6.2 is the input to the transformation and contains a root `specification` node with two children which are `actor` nodes. The XSLT in Listing 6.1 specifies a `template` to `match` any `specification` node in the input tree. The square brackets add an addition rule that the `specifcation` node must have at least one `actor` child for the match to succeed.

When a match occurs, the XSLT specifies that for each `actor` child, the transform must generate:

- the `name` of the actor, as specified by the `xsl:value-of` element,
- the literal text

  `: process (clk) begin end process;`

  as specified by the `xsl:text` element, and
- a newline character, as specified by the `
` escape sequence.

If we can write our source code (specification) in XML, then we can write our compiler in XSLT. Furthermore, if the compilation can be split into more than one step, we can write each part in a separate XSLT file, and use XML to represent the intermediate forms between steps. This allows us to partition the tasks associated with the transformation into logical units, each contained in a single stylesheet. Such modularity makes the system both

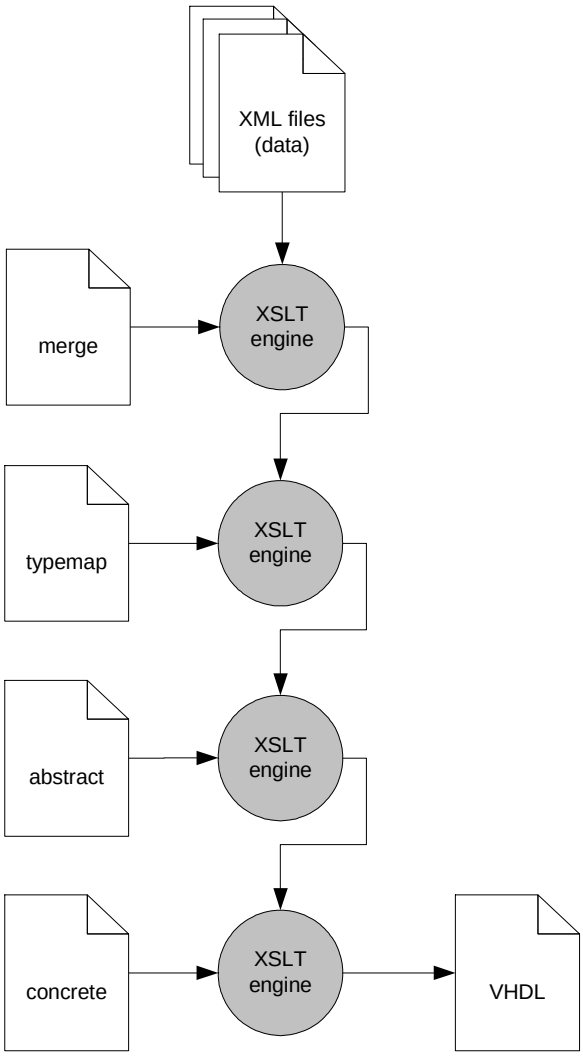


Figure 6.2: *Cascaded transformations for the autogeneration of VHDL.*

easier to understand and simpler to maintain, especially since the intermediate forms are now human-readable. The system which was implemented follows this approach, as shown in Figure 6.2.

One may wonder whether XSLT is powerful enough for the implementation of a compiler. The simple answer is that it must be, since it is a Turing-complete language [33], meaning that it can perform any computational task. However, this says nothing of whether it can perform our particular task in a way that is easy or efficient. The system presented below shows that given our requirements, the solution presented by XSLT is both easy and efficient.

We showed in Chapter 4 that the waveform specification can be represented as an XML tree with elements describing converters and links in a dataflow graph. To understand that the VHDL output can also be represented this way, consider that its structure can be represented as an AST. For example, a VHDL file (usually) contains an *entity* and an *architecture*. The VHDL file can be modelled as the root node of a tree, with these two structures as children.

A VHDL architecture may have *signals*, each of which can be seen as a child, and must have *body* which is also a child. The body may contain *processes* and they in turn may contain *variables*. Such a structure clearly has the properties of a tree, hence the name AST. In Chapter 5 we described which elements of VHDL are required to implement a waveform, a graphical depiction is provided in Figure 6.3.

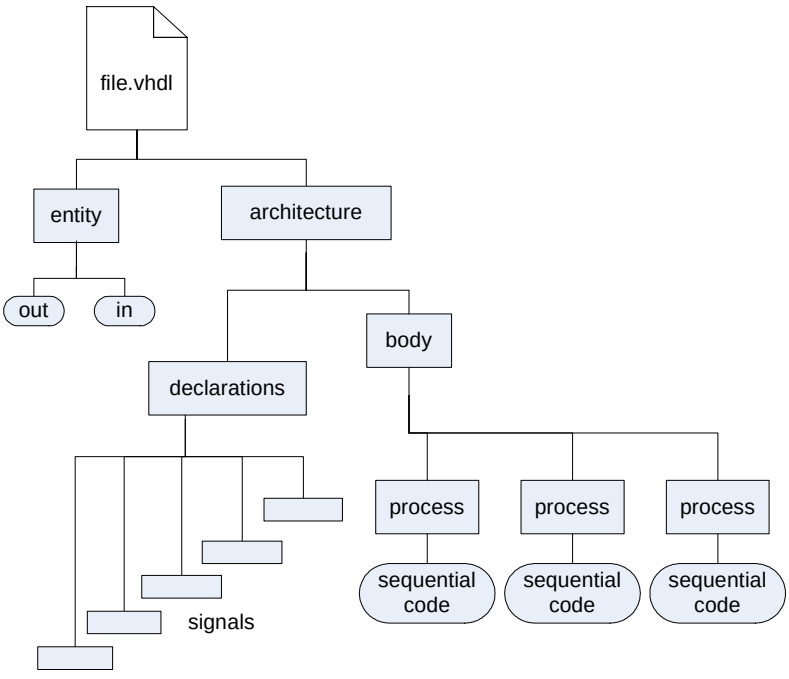


Figure 6.3: *VHDL abstract syntax tree for the autogenerated SDR.*

XSLT has two drawbacks which might be considered relevant when applying it to compiler

design or code generation. First, the input document to an XSLT processor must be well-formed XML. XSLT provides no support for converting documents from other forms into XML. This would be useful, for example, if one wanted to read in source code from some other language such as WDL (see Section 4.4) and convert to an abstract syntax tree in XML. However, such conversions are non-trivial as they depend on the grammar of the input, as described in Section 6.1. Because this parsing problem is considered to be outside the scope of XSL, support is not provided. However, given a formal language and its grammar, it is generally possible to construct such a front end using only syntax-directed translation [7]. This may be performed using any compiler-compiler program which is able to construct a parser for that grammar. Such a front-end is not required for our system because the specification is already in XML.

The second drawback of XSLT is that, since it is text oriented, it cannot generate binary output. This also does not affect our system, since our target languages (C++ and VHDL) are text based.

More importantly, the combination of XML and XSLT has the following advantages:

- The strict syntax of XML allows delegation of the parsing task to the XSLT processor.
- The intermediate forms are externally viewable, human readable, and can also be edited.
- XSLT is standardised [44], and many processors are available which support it.
- The extensibility of both XML and XSLT allow new ideas to be implemented quickly and easily.

Having motivated the design choice of XSLT, we now describe the transforms which were implemented for the VHDL target.

6.3 Transform system

The task was separated into four transformations which are detailed in this section. Each transformation is specified in an individual XSLT file. A build script is then responsible for transferring the output of one transformation to the input of the next, as shown in Figure 6.2. These four transforms constitute an original contribution of this work, and are summarised in Table 6.1.

The process is split into these four stages in order to achieve separation of concerns. This is desirable because it allows the design of any stage to be altered without affecting the other tasks. It is therefore possible, for example, to change the way in which components are resolved and fetched from a library (by modifying or replacing the *merge* transform) without

Table 6.1: *Summary of the four transforms and their associated tasks.*

Transform	Responsibility
Merge	Resolve component types and load missing XML specifications.
Typemap	Infer type information from links to component inputs and outputs.
Abstract	Map elements in the specification to the correct VHDL elements.
Concrete	Convert the abstract syntax to concrete VHDL.

affecting the mapping between the SDF structure and the VHDL AST (as specified in the *abstract* transform).

A description of each transform follows. For the full source code, refer to Appendix A.

6.3.1 *Merge* transform

The `merge` transform is the first transform to be applied to the waveform specification, and is applied to the top-level XML file. An example of such a top-level file was given in Listing 4.3 for the demodulation of FM by DPLL.

In this top-level file, `component` elements specify what functionality must be included in the algorithm by naming converters using the `type` attribute. For example, the XML segment in Listing 6.4 specifies that an `sdr_product` converter must be instantiated with the name `phase_comp` to perform the function of a phase comparator. The internal details of this converter are hidden from the author of this specification, since they are stored in a library. This library is simply a directory containing XML files for each type of converter.

Listing 6.4: *Component declaration before the merge transform.*

```
<component name="phase_comp"  type="sdr_product"/>
```

The information inside these files is essential for generating the correct VHDL code. Because the files are also in XML format, they can be combined with the input specification. The *merge* transform therefore writes all input nodes directly to the output, adding the necessary detail for each underspecified component by including the full XML tree for the corresponding converter definition below that component.

This is done using the identity transform, which copies all input nodes directly to the output, with one additional template to create an exception for `component` elements. For each node visited during the enumeration of the input tree, the XSLT processor looks for the first template in the XSLT file which matches that node. By introducing an extra template before the identity template, which matches only component nodes, we can cause these nodes to be handled in a special way.

Listing 6.5: *XSLT code for the merge transform.*

```
<xsl:template match="component[not(process)]">
        <xsl:copy>
            <xsl:apply-templates
                 select="*|@*|comment()|processing-instruction()|text()"/>
            <xsl:copy-of
                 select="document(concat('../../../generic/src/',@type,'.xml'))/*/*"
                 copy-namespaces="no"/>
        </xsl:copy>
        <xsl:apply-templates/>
</xsl:template>

<xsl:template match="*|@*|comment()|processing-instruction()|text()">
    <xsl:copy>
        <xsl:apply-templates select="*|@*|comment()|processing-instruction()|text()"/>
    </xsl:copy>
</xsl:template>
```

Due to the simplicity of this transform, it is fully reproduced in Listing 6.5. The second template implements the identity transform using recursion. The first template implements the exception described above, using the `document()` function to import additional XML information from file.

The component declaration of Listing 6.4 therefore becomes populated with child nodes which fully specify its behaviour, as shown in Listing 6.6

6.3.2 *Type-map* transform

A major difference between our μP and FPGA based platforms is that of data types, where we use floating- and fixed-point types respectively. This was discussed in Section 5.2. For μP platforms, floating-point data types can be used wherever real numbers must be represented, regardless of the range or precision required. In FPGAs, however, fixed-point numbers must be used, and each signal may require a unique word length and radix point location.

Table 6.2: *Attributes in the top-level specification which relate to data types.*

Attribute	Description	Possible values
`signed`	Specifies whether data type is signed or unsigned.	yes or no
`length`	Specifies word length of data type.	integer
`frac`	Specifies how many bits represent negative powers of 2.	integer

These properties are set in the top level specification as attributes of the `link` elements which specify the connections between converters in the system. The attributes which relate to type are described in Table 6.2. Note that these attributes are not assigned to the inputs

Listing 6.6: *Component declaration after the merge transform.*

```
<component name="phase_comp" type="sdr_product">
    <name>sdr_product</name>
    <author>R Brady</author>
    <date>2006-05-03</date>
    <description>Two input multiplier.</description>
    <input>
        <name>input1</name>
        <default>0.0</default>
    </input>
        .
        .
        .
    <process syntax="VHDL">
        <code>
            output := resize(input1 * input2, output);
        </code>
    </process>
</component>
```

and outputs of converter specifications. Consider the `sdr_product` converter specification which has been included by the *merge* transform as shown in Listing 6.6. The `input` element makes no reference to data type. Equally important is the observation that the VHDL code stub:

```
output := resize(input1 * input2, output);
```

makes no reference to data type either. The resize function used here further allows the inputs and output to have different word lengths

However, these inputs and outputs must be declared as variables in the generated code, and that declaration must specify a type. If we can generate that code correctly despite the lack of information, then the converter functionality is said to be *polymorphic* with respect to type. This is a powerful property which enables flexible re-use of the converter specification. It is made possible by the specification of data type information for each `link` element in the specification, as shown by the example in Listing 6.7.

It is the responsibility of the *typemap* transform to infer the correct type for each such input or output. This is done by determining which `link` element corresponds to each `input` and `output` , and then copying the relevant attributes so the type information is resolved. As with the *merge* transform, the XSLT consists of an identity transform with exceptions for the `input` and `output` elements which require additional information. The template for this exception is shown in Listing 6.8. The `select` attribute of the `xsl:for-each` element is used to locate the link which is connected to the input or output in question. The `xsl:attribute` element causes an attribute to be generated in the result tree, and this is

Listing 6.7: *Extract from the DPLL specification showning data type information as attributes.*

```
    <link signed="yes"
        length="16"
        frac="14"
        from="cos_lut"
        from_port="output"
        to="phase_comp"
        to_port="input1"/>
```

done for those attributes described in Table 6.2. The values for these attributes are then copied from the `link` element which has been found.

Listing 6.8: *Extract from the typemap transform showing the inference of type from links to converters.*

```
 <xsl:template match="converter/input | converter/output">
    <xsl:copy>
        <xsl:variable name="component" select="'self'"/>
        <xsl:variable name="port" select="./name"/>
        <xsl:for-each select="../process/link[(@from = $component and @from_port = $port) or
                                              (@to = $component and @to_port = $port)]
                                             [position() = 1]">
            <xsl:attribute name="signed"><xsl:value-of select="@signed"/></xsl:attribute>
            <xsl:attribute name="length"><xsl:value-of select="@length"/></xsl:attribute>
            <xsl:attribute name="frac"><xsl:value-of select="@frac"/></xsl:attribute>
        </xsl:for-each>
        <xsl:apply-templates select="*|@*|comment()|processing-instruction()|text()"/>
    </xsl:copy>
</xsl:template>
```

6.3.3 *Abstract* transform

This is an important transform which must map every concept in our specification document to an equivalent concept in VHDL. These conceptual mappings were discussed in Chapter 5. For example, for each `link` element in the input tree, a `signal` element must be created in the result tree, so that a VHDL signal is eventually declared in the generated code.

XSLT is a powerful language in this respect. In the example given by Listings 6.9 through 6.11, the XSLT not only matches the `link` element, creating a `signal` element in the result tree, but it also synthesises a unique name, `SDR_AUTOGEN_phase_comp_output` , from those attributes of the input `link` element which specify its endpoints. The lack of an `xsl:` prefix in the name of the `signal` element lets the XSLT engine know that this is an element which must be created in the result tree rather than a processing instruction of

the XSLT. The unique name for the newly created element is generated by concatenating several strings using the built-in `concat()` function. The `SDR_AUTOGEN` prefix is used to distinguish names in the final VHDL which are automatically generated, thereby preventing the erroneous redefinition of other identifiers, such as the names of input and output ports.

Listing 6.9: *XML element corresponding to a link between two converters.*

```
<link signed="yes"
      length="16"
      frac="14"
      from="phase_comp"
      from_port="output"
      to="loop_filter"
      to_port="input"/>
```

This transform does not have any specific responsibilities with respect to the data types of the various elements. These have already been resolved by the preceding *typemap* transform and the details are encoded in the attributes of the input tree, as can be seen in Listing 6.9. The type information is transferred directly to the output tree by copying the relevant attributes from the `link` element to the newly created `signal` element in the result tree. This is the purpose of the `xsl:copy-of` element in Listing 6.10.

Listing 6.10: *XSLT from the abstract transform specifying the mapping from a link in the specification XML to a signal declaration in the VHDL AST.*

```
<xsl:for-each select="process/link">
      <signal>
           <xsl:copy-of select="@signed | @length | @frac"/>
         <name>
           <xsl:value-of select="concat($autogen_prefix, @from, '_', @from_port)"/>
         </name>
      </signal>
</xsl:for-each>
```

Listing 6.11: *XML generated by applying the XSLT in Listing 6.10 to the XML in Listing 6.9.*

```
<signal signed="yes"
        length="16"
        frac="14">
    <name>SDR_AUTOGEN_phase_comp_output</name>
</signal>
```

The concepts presented in this example are applied to the entire structure, and the

result is a comprehensive transformation between the structure of the specification and the structure of the VHDL implementation. This mapping is depicted visually in Figure 6.4.

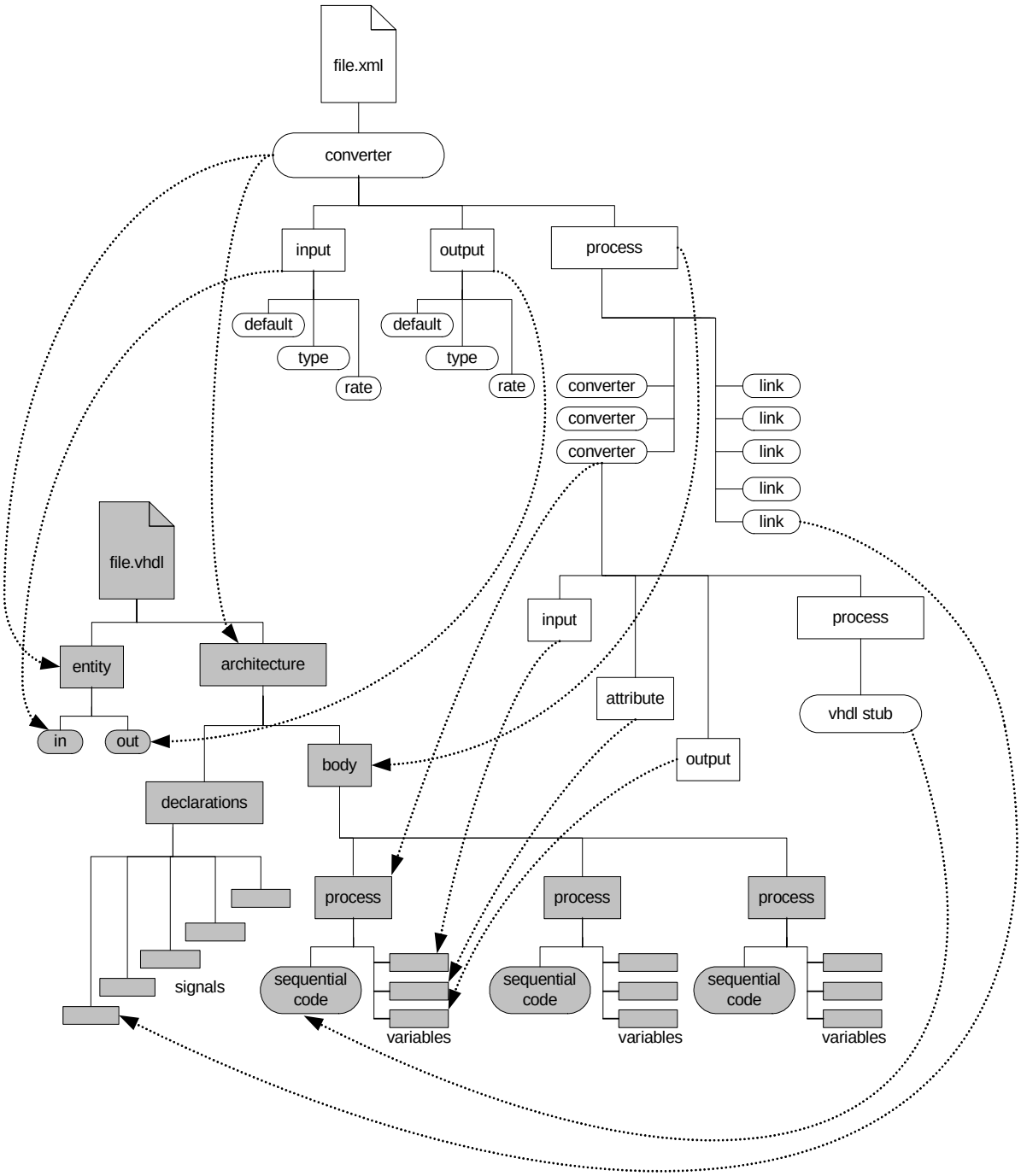


Figure 6.4: *Mapping from XML specification to VHDL AST.*

6.3.4 *Concrete* transform

VHDL is a formal language. It is important to produce VHDL code which conforms exactly with the standard so that it can be parsed by the synthesis tool being used. It is the job of

Listing 6.12: *Port declarations from the AST generated by the abstract transform.*

```
<port dir="input" type="std_ulogic">
  <name>clk</name>
</port>
<port dir="input" type="std_ulogic">
  <name>reset</name>
</port>
<port dir="input" type="sfixed(1 downto -14)">
  <name>input</name>
</port>
<port dir="output" type="sfixed(1 downto -14)">
  <name>output</name>
</port>
```

the *concrete* transform to generate actual VHDL code from the AST representation produced by the *abstract* transform. This is the only transform which does not output XML.

The task includes adding keywords, delimiters such as semicolons or parentheses, and whitespace to the output. Consider the XML in Listing 6.12 which was extracted from the output of the *abstract* transform. This XML shows the input, output, clock and reset ports of the top-level demodulation converter. As can be seen in the attributes for each `port` element, the types have been resolved so that each port will have the correct word length, radix-point location, and direction.

This information must be used to generate VHDL which declares two input ports and one output port. The XSLT code responsible for this is shown in Listing 6.13. The `xsl:text` element is used extensively for whitespace in order to make the generated code more readable.

Listing 6.13: *XSLT extract from the concrete transform for top-level port declarations.*

```
<xsl:if test="port">    PORT (&#10;</xsl:if>
<xsl:for-each select="port">
    <xsl:text>        </xsl:text><xsl:value-of select="name"/>
    <xsl:text> : </xsl:text>
    <xsl:choose>
        <xsl:when test="@dir='input'">IN  </xsl:when>
        <xsl:when test="@dir='output'">OUT </xsl:when>
    </xsl:choose>
    <xsl:value-of select="@type"/>
    <xsl:if test="position() != last()">
        <xsl:text>;</xsl:text>
    </xsl:if>
    <xsl:text>&#10;</xsl:text>
</xsl:for-each>
<xsl:if test="port">    );&#10;</xsl:if>
```

There is considerable subtlety in this code generation phase. For example, when declaring

input and output ports for an entity, each must be followed by a semicolon, except for the last. It is the responsibility of this transform to ensure such rules are adhered to. The transform therefore uses the

```
<xsl:if test="position() != last()">
```

test to check that the current port is not the last before writing a semicolon to the output with the

```
<xsl:text>;</xsl:text>
```

element. The resulting code is shown in Listing 6.14, and has the correct syntax as well as useful indentation to improve readability.

Listing 6.14: *Automatically generated VHDL for the ports in Listing 6.12.*

```
PORT (
        clk : IN  std_ulogic;
        reset : IN  std_ulogic;
        input : IN  sfixed(1 downto -14);
        output : OUT sfixed(1 downto -14)
    );
```

By following this approach for the full VHDL structure, the transform ensures that correct code is generated for all elements of the AST.

6.4 Conclusion

The system described above performs a complete transformation from XML specification to VHDL. The generated VHDL code implements the homogeneous SDF model of computation as a proof of concept. By modifying only the *abstract* transform, the system could be further developed to support the more general case of multirate SDF graphs using the handshaking techniques described in Chapter 5.

The generated VHDL code is synthesisable, and can therefore be processed by a FPGA synthesis tool such as Altera Quartus or Synplify Pro. These tools will generate a binary file which can be used to program (or reconfigure) the FPGA. We show in the next section that this results in correct execution of the the specified SDR waveform.

Chapter 7

System evaluation

In this chapter we evaluate the proposed system. We start by verifying the correctness of the generated code for a single converter, and then show results for demodulation of the FM waveform introduced in Chapter 3.

7.1 Synthesis results

To determine whether the generated code results in synthesis of correct logic, we specify an `sdr_product` converter for which each input and output port are connected to a `link` with two-bit word length. The autogenerated code for the converter is shown in Listing 7.1. This code was wrapped in an entity and architecture and passed to Altera Quartus for synthesis. The resulting logic as synthesised by Quartus is shown in Figure 7.1. In the figure, the following can be observed:

- the multiplication operation which takes the two inputs as operands,
- the `reset` signal driving multiplexers to select the correct output value upon reset, and
- the output register, as discussed in Section 5.1.4.

7.2 Reference waveform

We now consider the reference waveform introduced in Chapter 3. The specification consists of a top-level file, `fmrx.xml`, which instantiates several components, as shown in Figure 7.2. For the full specifcation, see Appendix A. Parameters in the specifcation were set for the reception of an FM signal with a center frequency of 2 MHz and a frequency deviation of 75 kHz, which is consistent with commercial radio broadcasting. The DPLL demodulates directly at the carrier frequency without down conversion.

Listing 7.1: *Autogenerated code for the sdr_product converter.*

```
    -- Main process with actor firing code
    phase_comp : PROCESS  (clk)
        VARIABLE input1 : sfixed(1 downto 0) := to_sfixed(0.0, 1, 0);
        VARIABLE input2 : sfixed(1 downto 0) := to_sfixed(0.0, 1, 0);
        VARIABLE output : sfixed(1 downto 0) := to_sfixed(0.0, 1, 0);
    BEGIN
      if rising_edge(clk) then
      if reset = '1' then
        output := to_sfixed(0.0, 1, 0);
        SDR_AUTOGEN_output_link <= output;
      else
        input1 := SDR_AUTOGEN_input1_link;
        input2 := SDR_AUTOGEN_input2_link;
        output := to_sfixed(0.0, 1, 0);
        -- Start actor code paste

    output := resize(input1 * input2, output);

         -- end actor code paste
        SDR_AUTOGEN_output_link <= output;
      end if;
      end if;
    END PROCESS;
END ARCHITECTURE behaviour;
```

The specification was then transformed using the system described in Chapter 6, which results in a single VHDL file. The specification was also used to generate C++ code using the previously available transformation system.

The resulting VHDL, along with a testbench which reads data from file and writes results to file, was compiled and simulated using ModelSim. The test input signal was obtained by using Matlab SimuLink.

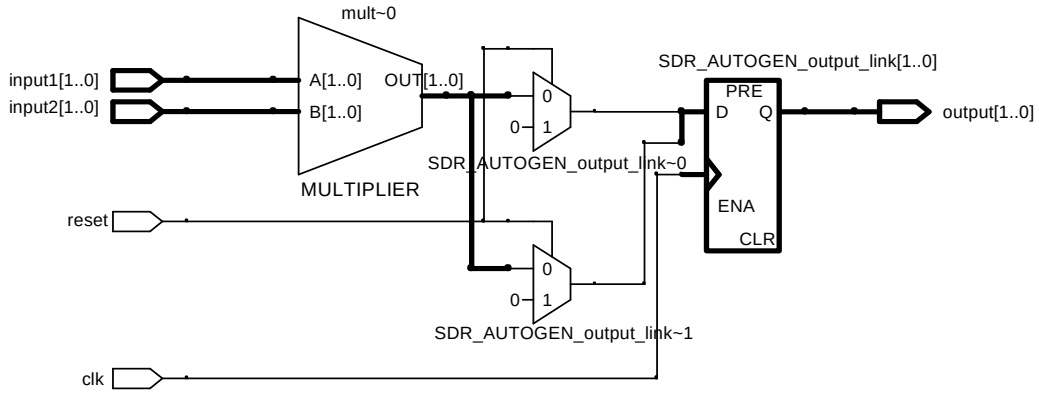


Figure 7.1: *Synthesised logic for the sdr_product converter.*

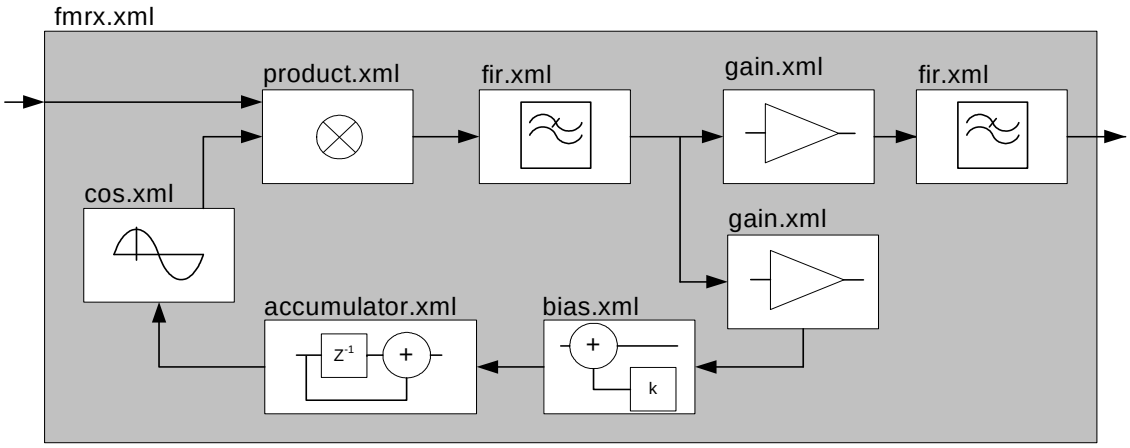


Figure 7.2: *FM demodulator described in XML.*

The results of both the C++ execution and the VHDL simulation are shown in Figure 7.3, which shows the demodulated signals. The small discrepancy between the two signals obtained on the two different platforms is attributed to the use of fixed-point data types in the FPGA implementation, where precision may have been lost.

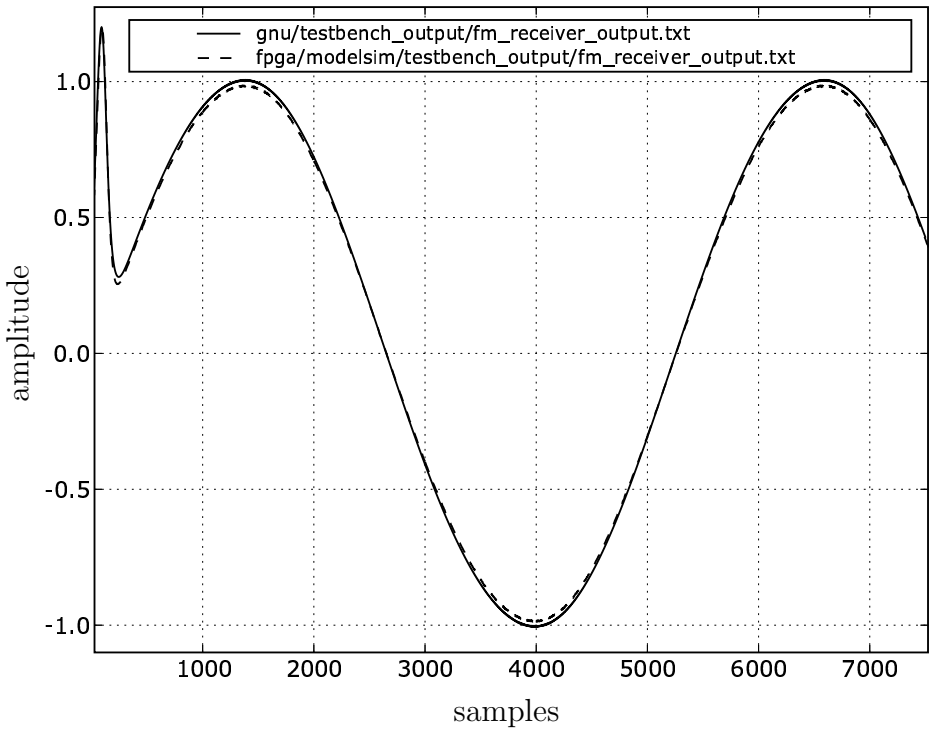


Figure 7.3: *Results for DPLL demodulation of test signal on both platforms.*

7.3 Reference platform

In this test, the FM waveform was targeted to the FPGA platform introduced in Chapter 3. The experimental test setup is pictured in Figure 7.4. Custom VHDL code was written to interface with the high-speed ADC at 25 MSps, and the audio DAC at 96 kSps.

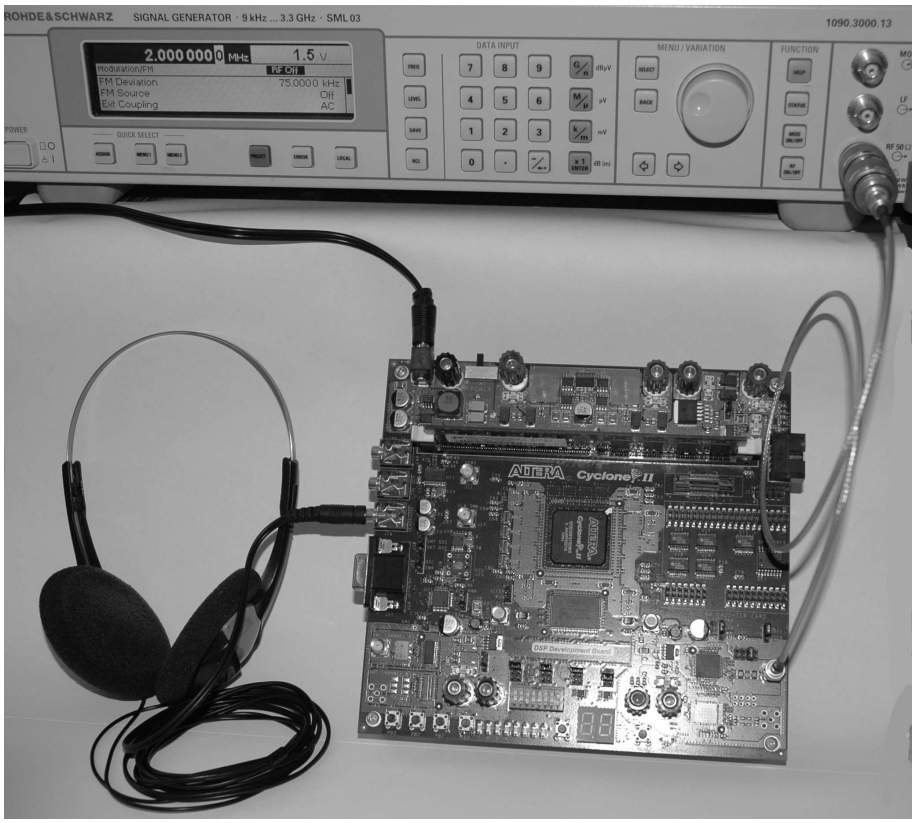


Figure 7.4: *Test setup for the demodulation of an FM signal using the Altera Cyclone II EP2C35 DSP development board as SDR platform and the Rohde & Schwarz SML03 signal generator as modulation source.*

The SML03 signal generator from Rohde & Schwarz was used as the signal source. It has the ability to generate its own modulating signal in the form of a single tone sinusoid. The frequency of this signal and the amplitude (with corresponding frequency deviation) can be set. A 5 kHz tone was used at an amplitude of 1 V to obtain a 2 MHz FM signal with a frequency deviation of 75 KHz. The original and demodulated signals are shown in Figure 7.5. Because the noise is visible on both the original and the demodulated signals, the bad quality of the signal is attributed to measurement effects and not the DPLL.

In the next test, the modulating signal was audio from a portable music player, which was input to the SML03 signal generator. The original and demodulated signals are shown in Figure 7.6. When listening to the demodulated output, the music could be heard loud and clear. It did, however, deteriorate rapidly when the centre frequency of the modulated

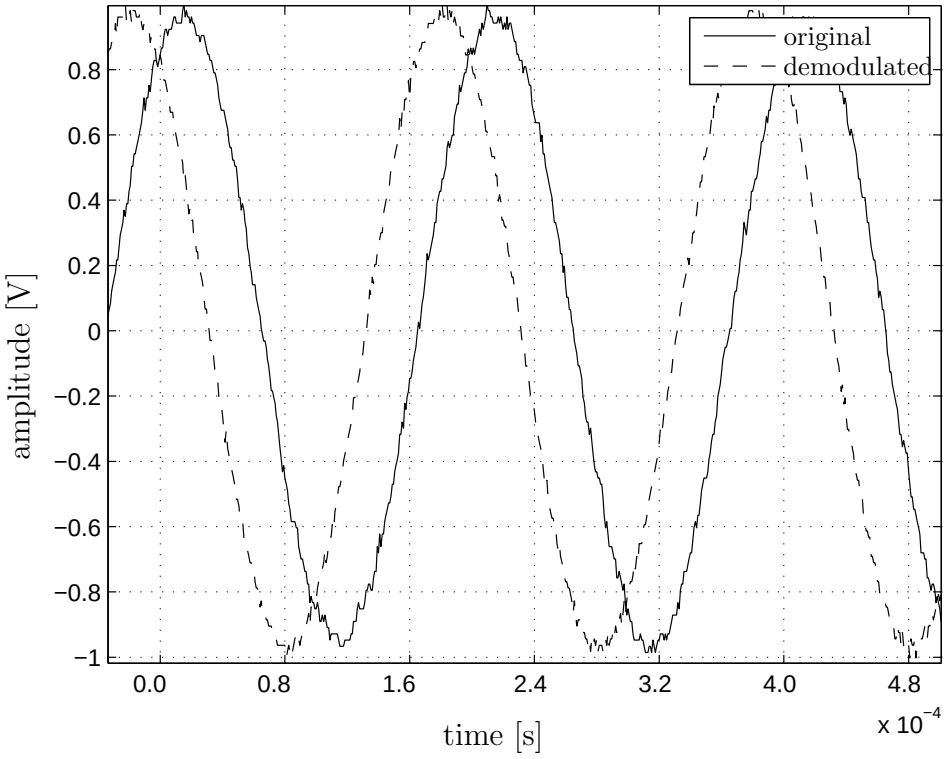


Figure 7.5: *Sinusoidal signal received be the FPGA based SDR (measurement noise is from the carrier signal, and has affected both the original and the received signals).*

signal was adjusted so that it did not match the demodulator. This is attributed to the simple low-pass filter design used, which was a FIR filter with only six taps.

7.4 FPGA resource usage

The FPGA resource usage, as reported by the Altera Quartus software, is reported in Table 7.1. Less than half the chip was used, with most of this being embedded multipliers.

The resulting VHDL code was also compared to a handcoded implementation. To avoid bias, a handcoded system was selected which had already been written by a third party. The synthesis results are compared in Table 7.2. They show that the autogenerated system uses up four times more area (logic resources) than the handcoded alternative.

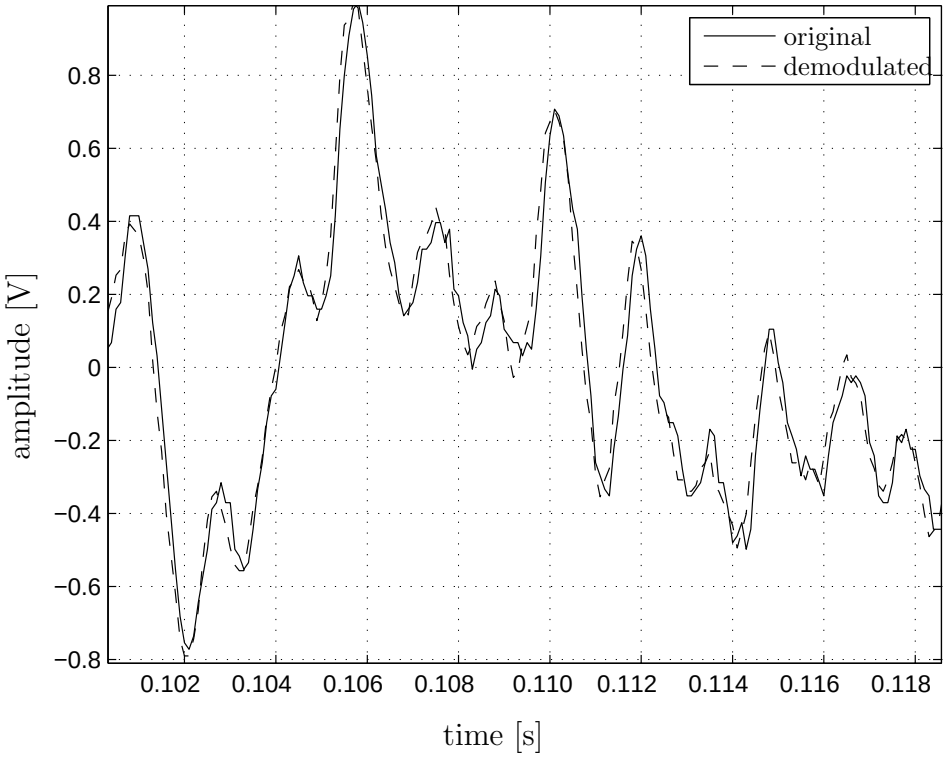


Figure 7.6: *Audio (music) signal received by the FPGA based SDR.*

Table 7.1: *FPGA resource usage for the reference design.*

Resource	Used	Total	Usage
Total logic elements	2,243	33,216	7 %
Total registers	598	33,216	2 %
Total LABs	166	2,076	8 %
I/O pins	43	475	9 %
M4Ks	4	105	4 %
Total memory bits	16,384	483,840	3 %
Embedded Multiplier 9-bit elements	34	70	49 %
PLLs	1	4	25 %
Global clocks	1	16	6 %

Table 7.2: *Comparison of hand-coded and autogenerated systems in terms of FPGA resource usage.*

Resource	Manually coded	Autogenerated
Logic elements (LEs)	707 /10570	2316 / 10570
LE usage	6%	21%
9-bit multiplier (DSPs)	0 / 6	1 / 6
DSP usage	0%	16%
Source lines of code (SLOC)	461	373

Chapter 8

Conclusion

As we move forward into the age of ubiquitous wireless communication, the number of waveforms used by various technologies shows no sign of slowing. To avoid obsolescence, terminals must support as many of these waveforms as possible. Ideally, a terminal should support even waveforms not known at design time.

Software defined radio provides a practical approach to achieving these goals. By transferring waveform support to the software domain, powerful reconfiguration is made possible. Implementing support for a waveform becomes a matter of writing the correct code for it.

When designing such a system, the choice of computation architecture is important. We have shown that microprocessors are not the only devices capable of SDR functionality, but that FPGAs also offer a powerful platform for this technology. Their ability to process data in a highly parallel way allows them to offer throughputs which are not yet achievable on microprocessor platforms.

However, as discussed in Chapter 2, this use of diverse platforms exacerbates the problem of diverse waveforms. Microprocessors and FPGAs have vastly different programming models, and waveform software must therefore be rewritten for each one. This is an expensive approach which can easily introduce errors and contradictions between different implementations of the same waveform.

In this thesis, we have shown that this is not necessary. By constructing an appropriate domain-specific language, we can specify our waveforms in a generic form. Code generation can then be used to obtain implementations for the different platforms.

In order to do this effectively, it is important to understand the nature of modulation and demodulation software. In Chapter 4 we show that such software conforms well to the synchronous dataflow model of computation, and use this to specify the structure of our modulation and demodulation algorithms. This is appropriate to software defined radio because many waveforms are simply different structural combinations of the same primitive elements. Examples of such primitive elements include FIR filters, multipliers or numerically controlled oscillators.

A contribution of this thesis has been the introduction of synchronous dataflow graphs

to provide us with a formal means to specify the connections and interactions between these primitives, in a way which is intuitive to the DSP engineer familiar with block diagram based systems. However, this is a semantic concept and requires a concrete syntax in order to be used. We write concrete specifications using XML with elements and attributes for all concepts which require specification.

This still leaves us with the challenge of specifying the internal behaviour of primitives. We note that sequential code with expressions and assignments is the most appropriate approach, and embed code stubs of C++ or VHDL inside our XML specifications. The similarity between C++ and VHDL at this level is far stronger than at the structural level, and syntax-directed translation of these code stubs is far more likely to succeed than with entire programs.

When autogenerating code from our XML specifications, we need to find appropriate structural frameworks in C++ and VHDL for the implementation of our SDF specifications. In C++, classes are used to represent converters, as was shown in previous work. A contribution of this thesis has been the development of a VHDL framework which provides a correct implementation of the SDF specification. This was described in Chapter 5.

The structured nature of this framework and the formal nature of our specifications enables us to autogenerate software implementations, as shown in Chapter 6. We do this using the XSLT language which applies template matching to our XML specification in several steps, eventually generating VHDL or C++. This approach has several advantages over traditional compiler design techniques when dealing with a domain specific language. It is not necessary to define a formal grammar, since this is provided by the markup of the XML in our specifications, and intermediate forms are both human and machine readable.

To demonstrate the principles above as well as the system which was developed, we introduced a reference waveform in Chapter 3. This FM waveform with a DPLL for demodulation was used throughout the thesis and provided a means to thoroughly test the system in Chapter 7. FM signals at radio frequencies with audio modulating signals were successfully captured and demodulated using an FPGA, demonstrating the practical use of the system which was developed.

In future work more complex waveforms would need to be considered. The logic resources for any particular FPGA may quickly become depleted when attempting to scale the system to such waveforms. This is a limitation of the current implementation. However, the development of techniques for resource sharing is an active research field, and existing techniques could be applied to the current system to alleviate the problem.

The result of this thesis has been a proof of concept for the implementation of SDR on FPGA-based platforms, while demonstrating that portability can be obtained by using formal models for specification in combination with code generation techniques. The proposed

system has the potential to reduce development effort and unify waveform specifications for the rapid deployment of new waveforms onto a wide range of reconfigurable SDR platforms.

Bibliography

[1] "The GNU Radio Project." http://www.gnu.org/software/gnuradio/, November 2006.

[2] "The Joint Tactical Radio Systems Project." http://jtrs.spawar.navy.mil/, nov 2006.

[3] "The XSLT Processor." http://xmlsoft.org/XSLT/, November 2006.

[4] "University of Stellenbosch Software Defined Radio Project." http://research.ee.sun.ac.za/sdr, November 2006.

[5] ABELSON, H. and SUSSMAN, G. J., *Structure and Interpretation of Computer Programs*. MIT Electrical Engineering and Computer Science, Second edition. The MIT Press, July 1996.

[6] AD, M., LAUWEREINS, R., and PEPERSTRAETE, J. A., "Hardware-software codesign with GRAPE." in *Rapid System Prototyping, 1995. Proceedings., Sixth IEEE International Workshop on*, pp. 40–47, 1995.

[7] AHO, A. V., SETHI, R., and ULLMAN, J. D., *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[8] ALTERA CORP., "Altera Cyclone II EP2C35 DSP Kit." http://www.altera.com/dsp, November 2006.

[9] ALTERA CORP., "Nios II Embedded Processor." http://www.altera.com/nios, November 2006.

[10] ATMEL CORP., "The AVR Microprocessor." http://www.atmel.com/avr, November 2006.

[11] BISHOP, D., "Fixed- and floating-point packages for VHDL-2005." Design and Verification Conference (DVCon), 2005.

[12] BLOSSOM, E., "GNU Radio: tools for exploring the radio frequency spectrum." *Linux J.*, June 2004, Vol. 2004, No. 122.

[13] BLUST, S., "Software Based Radio." in *Software Defined Radio: Enabling Technologies* (TUTTLEBEE, W. (Ed.)), Ch. 1, Chichester: John Wiley & Sons, 2002.

[14] BONSER, W., "US defense initiatives in software radio." in *Software Defined Radio: Origins, Drivers and International Perspectives* (TUTTLEBEE, W. (Ed.)), Ch. 2, Chichester: John Wiley & Sons, 2002.

[15] BOSE, V., ISMERT, M., WELBORN, M., and GUTTAG, J., "Virtual radios." *Selected Areas in Communications, IEEE Journal on*, 1999, Vol. 17, No. 4, pp. 591–602.

[16] BROOKS, C., LEE, E., LIU, X., NEUENDORFFER, S., ZHENG, H., and ZHAO, Y., "Introduction to Ptolemy II." in *UCB/ERL M05/21 Heterogeneous concurrent modeling and design in Java*, vol. 1, University of California at Berkeley, 2004.

[17] CAMPOSANO, R., SAUNDERS, L. F., and TABET, R. M., "VHDL as input for high-level synthesis." *Design & Test of Computers, IEEE*, 1991, Vol. 8, No. 1, pp. 43–49.

[18] CHAPIN, J., "Software Enigineering for Software Radios: Experiences at MIT and Vanu, Inc.." in *Software Defined Radio: Enabling Technologies* (TUTTLEBEE, W. (Ed.)), Ch. 10, Chichester: John Wiley & Sons, 2002.

[19] CHAPIN, J., LUM, V., and MUIR, S., "Experiences implementing GSM in RDL (the Vanu Radio Description Language)." in *Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. IEEE*, vol. 1, pp. 213–217 vol.1, 2001.

[20] CRONJÉ, J. J., "Software Architecture Design of a Software Defined Radio System." Master's thesis, University of Stellenbosch, 2004.

[21] CUMMINGS, M. and HARUYAMA, S., "FPGA in the software radio." *Communications Magazine, IEEE*, 1999, Vol. 37, No. 2, pp. 108–112.

[22] GALLOWAY, D., "The Transmogrifier C Hardware Description Language and Compiler for FPGAs." in *IEEE Symposium on FPGAs for Custom Computing Machines* (ATHANAS, P. and POCEK, K. L. (Eds)), (Los Alamitos, CA), pp. 136–144, IEEE Computer Society Press, 1995.

[23] GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J., *Design Patterns*. Addison-Wesley Professional, January 1995.

[24] GERBER, C., "Realtime scheduler for a software defined radio system." Final-year project, Department of Electrical and Electronic Engineering, Stellenbosch University, November 2006.

[25] GRELCK, C. and SCHOLZ, S.-B., "SAC: From high-level programming with arrays to efficient parallel execution." *Parallel processing letters*, 2003, Vol. 13, No. 3, pp. 401–412.

[26] HAMMES, J., RINKER, B., BOHM, W. A. P., NAJJAR, W. A., DRAPER, B. A., and BEVERIDGE, R. J., "Cameron: High level Language Compilation for Reconfigurable Systems." in *IEEE PACT*, pp. 236–244, 1999.

[27] HAREL, D., "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, June 1987, Vol. 8, No. 3, pp. 231–274.

[28] HAROLD, E. R., *XML Bible*. Gold edition. Hungry Minds, 2001.

[29] HORSTMANNSHOFF, J., GROTKER, T., and MEYR, H., "Mapping multirate dataflow to complex RT level hardware models." pp. 283–292, 1997.

[30] HUDAK, P., "Building domain-specific embedded languages." *ACM Computing Surveys*, 1996, Vol. 28, No. 4es.

[31] IEEE. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*, 2000.

[32] JUNG, H. and HA, S., "Hardware synthesis from coarse-grained dataflow specification for fast HW/SW cosynthesis." in *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 24–29, ACM Press, 2004.

[33] KEPSER, S., "A Simple Proof for the Turing-Completeness of XSLT and XQuery.." in *Proceedings of the Extreme Markup Languages 2004 Conference*, (Montral, Quebec, Canada), 2004.

[34] LEE, E. A. and MESSERSCHMITT, D. G., "Synchronous data flow." *Proceedings of the IEEE*, 1987, Vol. 75, No. 9, pp. 1235–1245.

[35] LEE, E. A. and PARKS, T. M., "Dataflow process networks." *Proceedings of the IEEE*, 1995, Vol. 83, No. 5, pp. 773–801.

[36] LEE, E. A. and MESSERSCHMITT, D. G., "Static scheduling of synchronous data flow programs for digital signal processing." *IEEE Transactions on Computing*, January 1987, Vol. 36, No. 1, pp. 24–35.

[37] MITOLA, J., "The software radio architecture." *Communications Magazine, IEEE*, 1995, Vol. 33, No. 5, pp. 26–38.

[38] PARR, T. J. and QUONG, R. W., "ANTLR: A Predicated-LL(k) Parser Generator." *Software Practice and Experience*, 1995, Vol. 25, No. 7, pp. 789–810.

[39] RUSHTON, A., *VHDL for Logic Synthesis*. New York: John Wiley & Sons, Inc., 1998.

[40] SHIRAZI, N., WALTERS, A., and ATHANAS, P., "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines." *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, 1995, pp. 155–162.

[41] SMITH, M. J. S., *Application Specific Integrated Circuits*. VLSI Systems Series. Addison-Wesley, 1997.

[42] TANENBAUM, A. S., *Structured computer organization (3rd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[43] TWITCHELL, E. R., "A digital approach to an FM exciter." *Broadcasting, IEEE Transactions on*, 1992, Vol. 38, No. 2, pp. 106–110.

[44] W3C CONSORTIUM, "XSL Transformations (XSLT) Version 1.0, W3C Recommendation." `http://www.w3.org/TR/xslt`, November 1999.

[45] W3C CONSORTIUM, "Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation." `http://www.w3.org/TR/xml`, August 2006.

[46] WILLIAMSON, M. C., "Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications." EECS Department, University of California, Berkeley, 1998.

[47] WILLINK, E., "The Waveform Description Language." in *Software Defined Radio: Enabling Technologies* (TUTTLEBEE, W. (Ed.)), Ch. 13, Chichester: John Wiley & Sons, 2002.

[48] ZIEMER, R. E. and TRANTER, W. H., *Principles of Communications: Systems, Modulation, and Noise*. Fifth edition. New York: John Wiley & Sons, Inc., 2002.

[49] ZIMMERMANN, H., "OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection." *IEEE Transactions on Communications*, 1980, Vol. 28, No. 4, pp. 425–432.

Appendix A

Source Code

Example source code for the designs discussed in this thesis is included on the accompanying CD. The directory structure on the disk is a subset of the libsdr environment. The directories and files of interest are listed in Table A.1.

Table A.1: *Contents of the accompanying CD.*

Location	Description
`libsdr/examples/fm_receiver`	location of top-level XML specification
`libsdr/examples/fm_receiver/...` `fpga/quartus/fm_receiver.vhd`	autogenerated VHDL file
`libsdr/platforms/generic/src`	generic converter library
`libsdr/platforms/fpga/src/xsl`	transforms for generating VHDL
`libsdr/platforms/gnu/src/xsl`	tranforms for generating C++
`libsdr/platforms/gnu/src/generated`	generated C++ files
`libsdr/platforms/gnu/include/sdr`	generated C++ headers